ARL-TR-8279 ● JAN 2018


US Army Research Laboratory

# A Practical Guide to the Open Standards for Unattended Sensors (OSUS)

by Jacob Tyo and Jesse Kovach

## NOTICES

### Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8279 ● JAN 2018

US Army Research Laboratory

# A Practical Guide to the Open Standards for Unattended Sensors (OSUS)

**by Jacob Tyo and Jesse Kovach**
*Sensors and Electron Devices Directorate, ARL*

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED (From - To) |
|---|---|---|
| January 2018 | Technical Report | 1 January 2018–31 December 2022 |

**4. TITLE AND SUBTITLE**
A Practical Guide to the Open Standards for Unattended Sensors (OSUS)

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Jacob Tyo and Jesse Kovach

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
US Army Research Laboratory
ATTN: RDRL-SES-A
Aberdeen Proving Ground, MD 21005-5066

**8. PERFORMING ORGANIZATION REPORT NUMBER**
ARL-TR-8279

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Open Standards for Unattended Sensors (OSUS) allows rapid device integration and interchangeability with larger systems through a standardized set of services that lets Open Source Gateway Initiative-compliant bundles execute on any OSUS platform. This report highlights some of the most important principles for proficiency in OSUS development and some of the benefits of implementing an OSUS controller as the central processing platform in a smart sensing device. The setup, development, and testing of OSUS plug-ins will be discussed through the analysis of 2 existing plug-ins and a walkthrough of a typical plug-in-development cycle.

**15. SUBJECT TERMS**
OSUS, Open Standard for Unattended Sensors, plug-in development, plug-in examples, integration, OSUS controller

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Jacob Tyo |
| Unclassified | Unclassified | Unclassified | UU | 91 | 19b. TELEPHONE NUMBER (Include area code) 301-394-1266 |

## Contents

## List of Figures



## List of Tables

INTENTIONALLY LEFT BLANK.

# 1. Introduction

Unattended ground sensors (UGSs) come in many shapes and sizes and often have many different components. These components range from infrared cameras and magnetometers to communications equipment and embedded computers. The combination of these different components requires sophisticated software that is often time consuming to develop and difficult to reuse. The US Army Research Laboratory (ARL) and the Defense Intelligence Agency created the Open Standards for Unattended Sensors (OSUS)[1] to simplify, streamline, and improve the reusability of UGS software. Based on OSGi,* OSUS allows for the seamless reuse of one component in multiple systems, significantly reduces development time, creates less complex source code, and results in a more maintainable software project. Furthermore, implementing an OSUS controller as the central processing platform of any smart sensing device allows for seamless reuse and replacement of underlying sensors, fast and simple mission reprogramming or repurposing, built-in communication between systems, and several methods of data exfiltration already implemented and available. This report is a practical guide that 1) provides a basic overview of the necessary programmatic background to understand OSUS and the underlying OSGi functionality, 2) presents examples to solidify important concepts, and 3) gives a walkthrough of a typical plug-in development's life cycle.

The reference documentation and source code are located at the following links:

- https://github.com/ssg-udri/OSUS-R/releases
- https://github.com/ssg-udri/OSUS-R

For those with a Common Access Card or Personal Identity Verification, more OSUS material can be found at these links:

- https://confluence.di2e.net/display/OSUS/OSUS+Home
- https://bitbucket.di2e.net/projects/OSUS/repos/osus-r/browse

This guide provides a walkthrough of 3 plug-ins: FakeTripwire, Sample Consumer, and Edge Detector. FakeTripwire (com.acme.assets.simple FakeTripwire) is an asset that randomly generates OSUS detections, similar to that of a camera or other sensor. Sample Consumer (com.acme.sampleConsumer) is able to receive persisted observations, and the Edge Detector (mil.arl.alg.edgeDetector) is able to receive persisted images, process them, and then persist the resulting image. For best

* The Open Source Gateway Initiative (OSGi) specification implements and describes a complete, dynamic, modular component and service model for the Java programming language.[2,3]

results, have a copy of these projects for reference while proceeding through this guide, as they will be heavily referenced. The average time needed to work though this document as intended is less than 1 day.

## 1.1 The OSUS Advantage

Using OSUS as the core of any smart sensing device includes all of the benefits mentioned in the introduction, as well as providing access to all of the previous work implemented in OSUS. Specifically, things such as radio- or satellite-communication plug-ins (modular functionality expansion parts of OSUS) are available for use and give the developer the ability to use these communication means by simply creating a configuration file. If multiple systems are in use, OSUS gives seamless communication between the units. Changing the application of the system is fast and simple with the different methods OSUS makes available for programming specific missions or objectives into the system. Lastly, any sensor that has an OSUS plug-in can be swapped into the system seamlessly. For example, if a camera becomes available with higher resolution than the one currently in use, swapping this into an OSUS system requires little (if any) code change to take advantage of the new camera.

## 1.2 This Guide vs. Other OSUS Documentation

This report is more basic than most other available OSUS documentation and more of a "getting started guide". It will heavily reference the other OSUS documentation and attempt to guide you in finding more specific details if necessary. Table 1 summarizes each of the currently available OSUS documents.

**Table 1 Comparison of OSUS documentation**

| Document | Description |
|---|---|
| OSUS plug-in guide[4] | Contains a vast amount of specific information on standard plug-in development, but requires background knowledge of OSGi functionality to fully comprehend |
| OSUS standard[1] | A detailed description of the OSUS standard and good reference. |
| OSUS mission programming guide[5] | Details how to configure OSUS for missions—out of the scope of this report. |
| OSUS-R operator instructions[6] | A guide on how to install, configure, operate, and interact with OSUS |
| OSUS remote interface specification[7] | A defined protocol for interacting with an OSUS controller—out of the scope of this document |
| *A Practical Guide to the Open Standard for Unattended Sensors (OSUS)* (this report) | Basics of OSGi and OSUS, provides definitions and an introduction by example |

## 1.3 The OSUS Controller

At a very high level, OSUS comprises a controller and plug-ins. The controller is the core piece of software that provides a means of communication and management for plug-ins. Plug-ins are added to the controller to extend the system's functionality.

A running version of OSUS is needed for this tutorial. For information on how to install, launch, and interact with OSUS, see Sections 3, 5.2, and 6.2 of *OSUS Operator Instructions.*[6]

## 1.4 Preparing an Integrated Development Environment (IDE)

OSUS plug-in development can be done from any IDE; however, Eclipse is highly recommended. For a more detailed description of setting up eclipse, getting the necessary add-ons (bndtools—the tools needed to build OSGi bundles), and using the OSUS software development kit (SDK), see Sections 5.1, 5.2, and 5.4 of the *OSUS Plug-in Guide.*[4]

As OSUS plug-ins become more complicated, having an OSUS environment within Eclipse that allows for debugging becomes increasingly important. Follow the instructions found in Section 5.9 of the plug-in guide[4] to set up an OSUS controller within Eclipse that will allow for normal debugging.

# 2. OSUS

OSUS was built on top of Apache Felix[8]—Apache's implementation of OSGi. OSUS uses this specification to ensure each system developed for OSUS will work with any other system. The OSUS Standard was then built on top of this, including a data model that allows for intelligent interactions among different parts of the OSUS system, specific to the goals and duties of sensors.

A plug-in and a bundle are the same thing. Plug-in is the typical OSUS terminology, and bundle is the typical OSGi terminology.[*] This document will use plug-in and bundle interchangeably. A plug-in is a jar file that contains all of the necessary methods to interact with OSGi (and, by extension, OSUS). Plug-ins are the modular part of the system, consuming and providing services that allow any OSUS controller to communicate with the plug-in or its underlying device.

[*] This is for historical reasons. Early iterations of the project that became OSUS were based on a custom module framework that used the term "plugin". When the system was redesigned to use OSGi, the old terminology remained in use.

In order to get such a modular piece of code to work with the rest of the system, specific methods, services, and protocols must be followed. The following sections will step through the most important elements needed for proper plug-in operations. Later sections will provide a walkthrough of existing plug-ins to show the proper implementation of these methods. The information found here represents the minimum knowledge needed to create OSUS plug-ins proficiently.

## 2.1 Plug-in Types and the OSUS SDK

Several different types of OSUS plug-ins exist, but they all fall into one of 3 categories: Asset, Communication, or Extension. Asset plug-ins are used for sensors and do things such as capture data, execute commands, and post observations. Communication plug-ins are used to provide physical, link, or transport layer support. Asset plugs often use a communication plug-in as a middleman between the asset and the OSUS controller. Lastly, extension plug-ins are used for anything else. This can include, but it not limited to, algorithms, fusion, and exfiltration. Figure 1 is a good graphical representation of the different types of plug-ins.

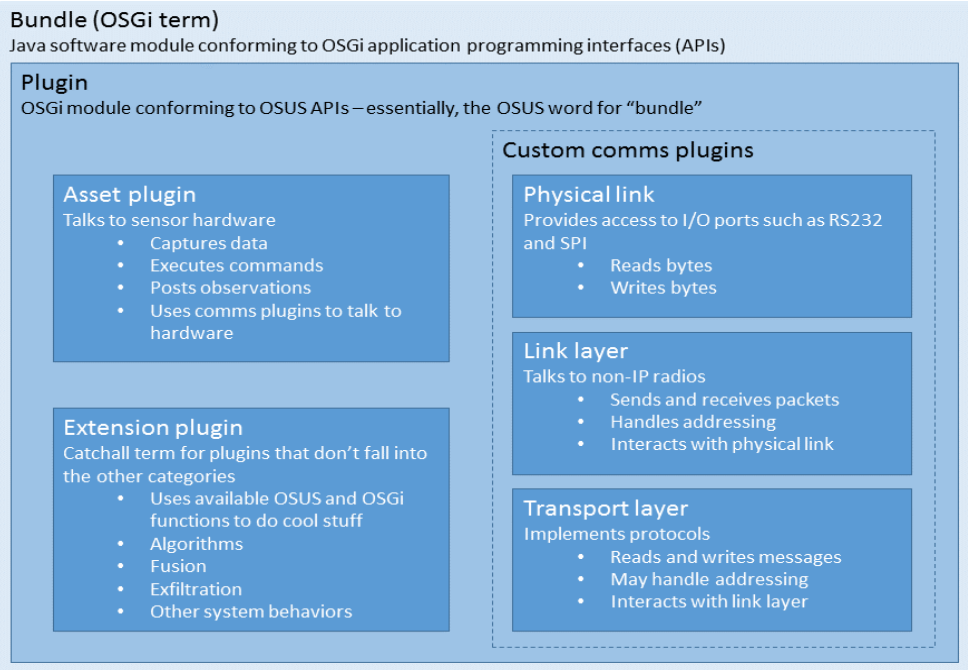


**Fig. 1 Different types of plug-ins**

Templates for different plug-in types are found in the OSUS SDK located here: https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/sdk-app-eneric.zip. For more information on the SDK, including how to create plug-in templates, see Section 5.1 of the OSUS Plug-in Guide.

## 2.2 Declarative Services and Annotations

The backbone of getting the proper operation from OSUS is OSGi declarative services and annotations. OSUS defines a number of standard services that can be provided or consumed by plug-ins. Plug-ins use annotations to indicate which services that plug-in provides or consumes. The build system (bndtools) reads these annotations to generate a declarative services Extensible Markup Language (XML) file, which is used internally by the OSGi runtime to link the appropriate services together as plug-ins are loaded. More information can be found at http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin/apache-felix-maven-scr-plugin-use.html and https://osgi.org/download/r5/osgi.cmpn-5.0.0.pdf.

There is a set list of annotations that must be used to properly create a plug-in and to extend its functionality to other parts of the system. These annotations (such as @Component or @Activate) will be discussed in Section 2.6 and during the walkthroughs.

## 2.3 Log Services, Event Admin, and Configuration Admin

The OSGi implementation that OSUS uses is Apache Felix. As part of this implementation, some Log services are provided along with an Event Admin and a Configuration Admin. The Log services provided are relatively standard and are not described here, but examples can be seen in the walkthroughs.

Event Admin provides a means for interservice and interplug-in communication, allowing plug-ins to distribute their events throughout the system or to receive events from other plug-ins. The walkthrough will describe the use and functionality of the Event Admin. If more information on the Event Admin is needed, see the documentation found here: http://felix.apache.org/documentation/subprojects/apache-felix-event-admin.html.

Configuration Admin is used to handle the configuration and attributes of each plug-in. Each plug-in denotes a specific set of variables that can be updated by the user or another part of the system; when changes are made to these variables, Configuration Admin makes them available to the plug-in. The OSUS implementation of this allows the configuration attributes to be modified from OSUS-SG, the standard graphical user interface (GUI). This provides a convenient way for an operator of the system to modify plug-ins.

## 2.4 Factories, Observations, and the Observation Store

Services within OSGi can be implemented as either standard objects or factory objects. There can be at most one instance of a standard object, whereas any number of instances of a factory object can be created. For example, asset plug-ins are implemented as factory objects, as multiple physical instances of an asset may be connected to a controller and one instance of the asset plug-in must be created for each connected physical asset. On the other hand, the observation store (described below) is implemented as a standard service as there is only one for the controller and all plug-ins.

"Observation" is an OSUS-defined object that can hold digital media (images, video, audio files, etc.), metadata about the capturing device and the media itself, detections, or status messages. Most OSUS media and data are stored and transferred in the form of observations. These observations are persisted with the observation store, and EventAdmin is used to notify other plug-ins of new observations as they are posted to the store. The observation store is a storage mechanism for observations that can be queried for any observation at any time from any component (as long as it has a reference to it).

The OSUS observation data structures are defined as XML Schema Definition (XSD) files, but can also be represented as Java classes and Protocol Buffers messages (see https://developers.google.com/protocol-buffers/ for more details on Protocol Buffers). Both of these representations are generated automatically from the XSD files,* and observations can be converted between any of these representations without loss of fidelity. Generally, the controller handles observations as Java classes, converting them to XML for validation and to Protocol Buffers for transmission over a network. OSUS performs schema validation on persisted observations. The best way to ensure that the persisted observation is valid, or to see exactly what objects are supported, is by looking through the XML schema. The mil.dod.th.core.schema.zip file provided with the OSUS binary distribution contains the schema files, or they can be found at the following location within the controller source code:

osus-r\mil.dod.th.core.lexicon\schemas\core\observation\types\

The top-level schema definition for Observation objects can be found in the Observation.xsd file.

* The Java classes are generated using JAXB. The Protocol Buffers' definitions are generated from the JAXB annotated classes using custom tools. These tools are included with the OSUS source distribution.

## 2.5 Attributes and Configuration Management

Each plug-in has an attributes file. This file is used to declare variables, which can be edited by other parts of the system, including an operator through OSUS-SG (Standard GUI). This will be shown and discussed in more detail in the walkthrough.

Assets have a capabilities-xml file. This file details the capabilities of each plug-in in terms of the functionality it provides. A plug-in may not behave properly if this file is not properly set up. An example of this will be shown in the FakeTripwire walkthough. The XSD files defining the expected capabilities file can be found at

osus-sdk\mil.dod.th.core.lexicon\schemas\core\asset\capabilities\

and

osus-sdk\mil.dod.th.core.lexicon\schemas\core\capability\.

## 2.6 Plug-in Life Cycle

The high-level life cycle of an OSGI bundle is activation, modification, and deactivation.

When all of a bundle's dependencies are met, the activate method is called to initialize the services provided by that bundle. If at any time the bundle's configuration is changed, the modified method is called to update the configurations appropriately. When the bundle is stopped, or when its dependencies are no longer met, the deactivate method is called to shut down the bundle's services.

OSUS assets and communications plug-ins have an additional activate–deactivate life cycle that exists side by side with the OSGI activation–deactivation life cycle.[*] Asset activation and deactivation occurs on command (initiated by either a user or another plug-in), in contrast with bundle activation that occurs automatically whenever a bundle is loaded and its dependencies are satisfied. The extended life cycle is initialization (corresponding to bundle activation), asset activation, modification, asset deactivation, and bundle deactivation. Asset activation and deactivation may occur multiple times during a component's lifespan.

[*] This highly unfortunate terminology conflict exists for historical reasons. As previously noted, early versions of OSUS were based on a custom module framework, and the terms "activate" and "deactivate" were adopted for use within this custom framework. When the custom framework was replaced with OSGi, the old terms were not changed and the conflict with the identically named but semantically different OSGi terminology was introduced.

By contrast, extension plug-ins interact directly with the OSGi-bundle life cycle. These plug-ins provide their own activate, modified, and deactivate methods that must be annotated with @Activate, @Modified, and @Deactivate annotations.[2,9,10] (These annotations must *not* be used on asset or communications plug-ins or unexpected behavior will occur.)

Table 2 shows the comparison of OSGi and OSUS method names and a description of each step. This will be revisited in the walkthroughs.

**Table 2 Comparison of OSGi and OSUS life-cycle terminology**

| OSGi annotation | OSGi method | OSUS asset method | Description |
|---|---|---|---|
| **@activate** | Activate | Initialize | Creates the plug-in instance. |
| . . . | . . . | onActivate | OSUS specific; sets the plug-in ready to capture, generate, or receive data (running state). |
| **@modified** | Modified | Updated | Informs the plug-in of configuration changes. |
| . . . | . . . | onDeactivate | OSUS specific; sets the plug-in *not* ready to capture, generate, or receive data (paused state). |
| **@deactivate** | Deactivate | . . . | Shuts down the plug-in when it is stopped or its dependencies are no longer met. |

## 2.7 Mission Programming

This mission programming of OSUS is out of the scope of this paper; however, the *OSUS Mission Programming Guide*[5] provides excellent detail on this topic. The JavaScript method described in the aforementioned document is not the only method of OSUS mission programming. Another common method is to create a plug-in that acts as the mission program. An example would be a plug-in that waits for a specific detection and then triggers the camera to take a picture. Both methods can be made to accomplish any task and be easily swapped for a new mission program or objective.

# 3. Plug-in Walkthrough

This section of the report provides an analysis and discussion of 2 existing plug-ins to show the typical composition of both a plug-in that produces data (such as a camera) and a plug-in that consumes data (such as an algorithm). This section concludes with a discussion of the development process of a plug-in that both consumes and produces data. This in combination with the previously defined OSUS and OSGi terminology will provide enough information to allow for the efficient and effective development of a typical OSUS plug-in.

## 3.1 simpleFakeTripwire

The simpleFakeTripwire asset plug-in randomly generates detections for testing and example purposes. This section will analyze and discuss each of the java files associated with the simpleFakeTripwire asset, with emphasis on the previously defined OSUS specific components.

### 3.1.1 simpleFakeTripwireAsset.java

The simpleFakeTripwire is an asset and therefore used the asset template from the OSUS SDK. This template implements the `AssetProxy` interface, and provides some asset-specific functionality through the `AssetContext` class reference. The `@Override` annotation is marked above many methods in this file, but the needed annotations (such as `@Activate`, `@Modified`, and `@Deactivate`) are encapsulated in the implemented interface. This guide will point out these overridden annotations where necessary. The annotation and class declaration are as follows:

```
@Component(factory = Asset.FACTORY)
public class simpleFakeTripwireAsset implements AssetProxy {
```

The `@Component` annotation marks the class as a component and assigns its factory as the `Asset.FACTORY`. This specific factory assignment allows the OSUS controller to get lists of assets, based on their presence in this specific Factory; if creating an asset plug-in, this factory assignment must be present or the OSUS controller will not recognize the asset. Furthermore, this allows for the instantiation of multiple assets. To use the OSUS functionality properly, an asset must implement the `AssetProxy` interface.

```
    // property variables
    int interval;
    int statusInterval;
    SensingModalityEnum modality;

    // class variable for housing data generator
    DataGenerator dg = null;

     /**
     * Class constructor.
     */
    public simpleFakeTripwireAsset() {
        // super in this case calls java's object class constructor,
which
        does nothing
        super();

        // create a new DataGenerator
        dg = new DataGenerator();

    }
```

The class-variable declarations and class constructor are shown above. Three variables are defined for configuration management, and the `DataGenerator` is created to provide the example plug-in functionality.

```
    /**
     * Reference to the context which provides this class with
methods to
     * interact with the rest of the system.
     */
    private AssetContext m_Context;

    /**
     * initialize is called when all of the plug-in's
dependencies are met,
     * and it can be created
     */
    @Override
    public void initialize(final AssetContext context, final
Map<String,
    Object> props) throws FactoryException {
        // set the provided context to the class variable for
later use
        m_Context = context;
        // set the initial properties of this plug-in
        updated(props);

        // Set the plug-in's status message
        m_Context.setStatus(SummaryStatusEnum.OFF, "Asset
Inactive");
    }
```

The above code snippet shows the creation of the `AssetContext` class variable and the `initialize` method. The initialize method is the method corresponding to the `@Activate` annotation and therefore is called when the plug-in is being created (bundle activation). When the OSGi framework calls the initialize method, it provides a reference to the `AssetContext` that is used to set the class variable and therefore gain access to the OSUS-defined `AssetContext` methods.

```
     /**
     * updated is called when the properties of the plug-in are
changed and
     * need updated
     */
    @Override
    public void updated(final Map<String, Object> props) {
        // create a configurable from the input properties.  This
is used to
        set the user defined or modified properties to class
variables
        final simpleFakeTripwireAssetAttributes config =
            Configurable.createConfigurable(
```

```
            simpleFakeTripwireAssetAttributes.class, props);

        //set the class variables to the input properties
        interval = config.interval();
        statusInterval = config.statusInterval();
        modality = config.modality();

        // restart the data generation functionality
        dg.startOrStopTimer();
    }
```

The `updated` method, corresponding to the `@Modified` annotation, controls what is done when the plug-in configuration changes. Specifically, a `Configurable` object is created from the attributes defined in the corresponding asset-attribute file (`simpleFakeTripwireAssetAttributes.java`) and is used to set the class variables accordingly. Following the class-variable assignment, the data-generation functionality is started (out of scope of this report and will not be discussed). The asset-attribute file will be discussed in more detail later.

```
    /**
     * onActivate is called when the plug-in is to be activated
     */
    @Override
    public void onActivate() throws AssetException {
        // When the asset is activated, we will generate a random
detection
        // at periodic intervals
        // (delay is specified in the asset configuration) and post
it to the
        // persistent store.

        // set the data generator to active
        dg.active = true;
        // Start the timer that will generate the periodic image.
        dg.startOrStopTimer();

        // Log an activation method
        Logging.log(LogService.LOG_INFO,      "Fake      Tripwire
activated");

        // this will also generate/send a status observation
        m_Context.setStatus(SummaryStatusEnum.GOOD,         "Asset
Activated");
    }
```

As mentioned in the plug-in life-cycle section, asset plug-ins have an extra startup step (plug-in activation). This extra step is handled in the above method, `onActivate`. When the asset is set to begin generating data, the `onActivate` method is called and the data generator is set to active, the functionality is started, and some messages are logged.

```
    /**
```

```
     * onDeactivate is called when the asset is deactivated, and
therefore
     * must
     */
    @Override
    public void onDeactivate() throws AssetException {
        // Log a deactivation message
        Logging.log(LogService.LOG_INFO,      "Fake      Tripwire
deactivated");

        // Set variable to stop the timer
        dg.active = false;
        // Stop the timer.
        dg.startOrStopTimer();

        // Set the asset status accordingly
        m_Context.setStatus(SummaryStatusEnum.OFF,          "Asset
Deactivated");
    }
```

The `onDeactivate` method is defined in the `AssetProxy` interface and controls the asset deactivation (described in Table 2 as putting the plug-in in a pause like state – plug-in deactivation). This method stops the data generation functionality, and sets the status of the asset to deactivated. The `onDeactivate` method is different from a method bearing the `@Deactivate` annotation, as such a method would uninstall the plug-in from the controller instead of simply changing the status of the asset.

```
    /**
     * onCaptureData is called when the asset is to capture data.
This
     * method will call the
     * observation creation/capturing methods, and then return the
new
     * observation.
     */
    @Override
    public Observation onCaptureData() {
        // Generate and return a single observation. (The base class
will
        // handle posting the observation to the persistent store.)
        Observation obs = dg.generateObservation(null);

        // Log a message accordingly
        Logging.log(LogService.LOG_INFO,   "Fake   Tripwire   data
captured");

        // Return the generated observation
        return obs;
    }
```

The `onCaptureData` method is shown above, and defined in the `AssetProxy` interface. This asset specific method handles the data capturing functionality of the plug-in. The comments in the above code snippet give a good description of the functionality.

```
    /**
     * onPerformBit performs some type of self checking, and then
returns a
     * status with respect to health of the plug-in.
     */
    @Override
    public Status onPerformBit() {
        // onPerformBit should be a health checking measure,
however as this
        // plug-in is a data generator there is nothing to check
        // Log an appropriate message
        Logging.log(LogService.LOG_INFO, "Performing Fake Tripwire
BIT");
        // return a status representing the results of the self
check
        return new Status().withSummaryStatus(new
            OperatingStatus(SummaryStatusEnum.GOOD,          "BIT
Passed"));
    }
```

Some self-checking functionality is needed for assets, and is called from the `onPerformBit` method. Because this example is a data generator, there is nothing to check. But if this were an asset such as a camera, functionality would need implemented here that would allow the asset to provide a report of its state of health.

```
    @Override
    public      Response      onExecuteCommand(final      Command
capabilityCommand) throws
        CommandExecutionException {
             // No commands currently supported.
             throw new CommandExecutionException("This asset does
not support
                    any commands.");
    }
```

The last OSUS specific method in the simpleFakeTripwire Asset is the `onExecuteCommand` method. This method is not supported in this example, but exists to handle commands from the controller. (For more information, see the OSUS Plug-In Guide.) Although this plug-in persists observations, the specifics of this will be covered in the edgeDetector walkthrough.

### 3.1.2 simpleFakeTripwireAssetAttributes.java

OSUS defines the `AssetAttributes` class for configuration management. Extending this class into a plug-in makes attribute management easy. This small class determines what attributes can be changed while the plug-in is running and also abstracts the parsing and processing requirement of getting these attributes into the plug-in.

```
/**
 * Interface which defines the configurable properties for a
 * simpleFakeTripwire.
 */
@OCD(description                                                    =
ConfigurationConstants.PARTIAL_OBJECT_CLASS_DEFINITION)
public    interface    simpleFakeTripwireAssetAttributes    extends
AssetAttributes {
    /**
     * Each method annotated with @AD becomes a configuration
property
     * available
     * to the plug-in. The return type of each method is the type
of the
     * configuration property. All simple types are supported and
other types
     * are supported if they can be converted from a string (e.g.,
class with
     * a constructor accepting a string or an enum). Also, the type
can be an
     * array or a collection for properties with multiple values.
     */

    @AD(required  =  false,  deflt  =  "60000",  name  =  "Detection
Interval",
        description    =    "Detection    Generation    Interval
(milliseconds)")
    int interval();

    @AD(required  =  false,  deflt  =  "300000",  name  =  "Status
Interval",
        description = "Status Generation Interval (milliseconds)")
    int statusInterval();

    @AD(required  =  false,  deflt  =  "PIR",  name  =  "Modality",
description =
        "Modaility (from SensingModalityEnum)")
    SensingModalityEnum modality();

}
```

The `@AD` annotation represents an Attribute Definition, and a variable declared as shown above will be accessible to the plug-in as well as users and other plug-ins. Referencing back to the `updated` method previously discussed in the simpleFakeTripwireAsset.java file, every variable defined in

`simpleFakeTripwireAssetAttributes` is made accessible through a configurable.

### 3.1.3 simpleFakeTripwire Capabilities—com.acme.assets.simpleFakeTripwire.simpleFakeTripwireAsset.xml

The following capabilities file describes the data produced by the simple FakeTripwire as well as the commands it supports. This file must be names packageName.className.xml, or it will not be found by the system. As shown on the next block of script, the `<ns2:modalities description="none" value="Imager"/>` line describes the asset as an imager. Some range and field of view capabilities are then specified, followed by the detection capabilities. This capability file concludes with a list of OSUS commands that the plug-in supports. The best way to see all of the possible fields and values that can exist in this file is by looking at the one generated by the OSUS SDK when creating a new project. The newly created capabilities file will contain all possible fields and example values. The XSD files describing these files, which are the files the capabilities are validated against, can be found at the locations specified in Section 2.5.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:AssetCapabilities          xmlns="capability.core.th.dod.mil"
xmlns:ns2="capability.asset.core.th.dod.mil"
xmlns:ns3="capability.link.ccom.core.th.dod.mil"
xmlns:ns4="capability.transport.ccom.core.th.dod.mil"
xmlns:ns5="capability.physical.ccom.core.th.dod.mil"
manufacturer="Acme Corporation, Tactical Systems Division"
description="Generates random detections of various types"
productName="Acme Fake Tripwire">
    <primaryImage encoding="image/jpeg">/9j</primaryImage>
    <secondaryImages encoding="image/jpeg">/9j</secondaryImages>
    <ns2:modalities description="none" value="Imager"/>
    <ns2:minRange>0.0</ns2:minRange>
    <ns2:maxRange>20.0</ns2:maxRange>
    <ns2:nominalRange>20.0</ns2:nominalRange>
    <ns2:minFov>20.0</ns2:minFov>
    <ns2:maxFov>20.0</ns2:maxFov>
    <ns2:nominalFov>20.0</ns2:nominalFov>
    <ns2:detectionCapabilities                  targetId="false"
directionOfTravel="false"                  trackHistory="false"
targetFrequency="false"                    targetLOB="false"
targetOrientation="false" targetRange="false" targetSpeed="false"
targetLocation="false">
        <ns2:typesAvailable>Alarm</ns2:typesAvailable>
        <ns2:typesAvailable>Test</ns2:typesAvailable>
        <ns2:classifications value="Other" />
    </ns2:detectionCapabilities>
    <ns2:commandCapabilities performBIT="true" captureData="true">
```

```
<ns2:supportedCommands>GetPositionCommand</ns2:supportedCommands>

<ns2:supportedCommands>GetVersionCommand</ns2:supportedCommands>

<ns2:supportedCommands>SetPositionCommand</ns2:supportedCommands>
    </ns2:commandCapabilities>
</ns2:AssetCapabilities>
```

### 3.1.4 simpleFakeTripwireAssetScanner.java

If supported, this file controls how the plug-in scans for connected assets. This is out of the scope of this report. For more information, see the *OSUS Plug-in Guide*.[4]

### 3.1.5 Summary of the FakeTripwire Asset

The FakeTripwire Asset was developed as an example plug-in with mock functionality to show the typical composition of an OSUS plug-in. This mock asset demonstrates basic functionality of the mandatory OSUS functions with emphasis on `initialize`, `updated`, `onActivate`, `onDeactivate`, and `onCaptureData`. The `initialize` method is executed when the plug-in dependencies are satisfied (bundle activation), whereas the `onActivate` method is executed when the asset is activated (plug-in activation). The `onDeactivate` method is executed to deactivate the asset into a pause-like state (plug-in deactivation), and the `updated` method controls how configuration (attribute) changes are handled. The onCaptureData is executed when the plug-in is signaled to collect or generate data, and lastly, modifying the plug-in attributes during runtime is made easy with the AssetAttributes class and a configurable object.

## 3.2 SampleConsumer

The sampleConsumer plug-in was created as an example of how to gather observations from the observation store. The nontrivial point in this class is how the observations must be gathered, because if not done on a separate thread the operation times out. Not all of the previously mentioned OSUS-specific points in the FakeTripwire walkthrough will be repeated, but all new material will be discussed.

### 3.2.1 Consumer.java

The Consumer as stated above was created to gather observations from the observation store. This functionality is not as abstracted as persisting an observation (such as in FakeTripwire), and therefore more must be set up manually.

```
@Component(
```

```
        // provides EventHandler service to receive OSGi events
        provide={EventHandler.class},
        // activate always even if no consumers
        immediate=true,
        // class containing config info for the metatype and config
admin
        // services
        designate=ConsumerConfigInterface.class,
        // activate bundle even if configuration does not exist
        configurationPolicy=ConfigurationPolicy.optional,
        // register for events on this topic
        properties={EventConstants.EVENT_TOPIC + "=" +
            ObservationStore.TOPIC_OBSERVATION_PERSISTED
            + "|" + ObservationStore.TOPIC_OBSERVATION_MERGED,
    } )
    public class Consumer implements EventHandler {
```

To properly prepare a class to observe system events (such as when another plug-in persists an observation), the `@Component` annotation must be handled a little differently than what was seen for an asset (FakeTripwire). The `provide={EventHandler.class}` argument provides the Event Handler service to allow for the reception of events. The `immediate=true` parameter is needed, because OSGi only creates plug-ins that something else depends on. Often with plug-ins of this nature, nothing will depend on it and therefore nothing will be created; however, the `immediate` argument will ensure the plug-in is initialized, even if there are no consumers of its service. The configuration parameter sets the presence of a configuration optional. The last parameter, `properties`, registers the plug-in to receive events on a specific topic. This functionality can be used as a filter in the case there is only a specific event topic under concern. The `EventHandler` is then implemented, as this is the main interface for receiving events.

```
    Boolean m_run = false;
    //  queue holding UUIDs to be processed by the logger thread
    BlockingQueue<UUID>         m_eventQueue         =         new
LinkedBlockingQueue<UUID>();
    EventProcThread m_eventprocessor = null;
    ObservationStore m_obsStore = null;
```

Several class variables are defined to later assist with keeping track of things such as if the consumer is active and if there are a separate thread for processing events, a queue to keep track of the events, and a reference to the observation store.

```
    @Reference

    // get reference to the ObservationStore service so we can
retrieve
    // observations after they are posted
    // This method is called by the framework due to the @Reference
    // annotation.
```

```
    public void setObservationStore(ObservationStore obsStore)
    {
        m_obsStore = obsStore;
    }
```

The `@Reference` annotation represents a dependency. This dependency calls for a reference to the observation store using the external `setObservationStore` method. Furthermore, the OSGi runtime calls this method during initialization and passes it the reference to the observation store that exists elsewhere in the system. If this reference cannot be satisfied (e.g., an observation store does not exist), the plug-in will not be created.

```
    @Activate // <- tells bnd this is the activate method
    // activate method called by the framework when all dependences
have been
    // satisfied and the bundle should start processing
    public void activate(Map<String, Object> properties)
    {
        updateConfig(properties);
        init();
    }
```

FakeTripwire implemented the `AssetProxy`, which abstracted the OSGi annotations. This plug-in does not have that convenience and therefore the `@Activate`, `@Modified`, and `@Deactivate` annotations must be handled manually. The `activate` method here is equivalent to the the `initialize` method in the `AssetProxy` and is called during bundle activation.

```
    @Deactivate // <- tells bnd this is the deactivate method
    // deactivate method called by the framework when the bundle
should be
    // shut down
    //  because  the  framework  is  stopping/the  bundle  is  being
uninstalled/etc.
    public void deactivate()
    {
        stop();
    }
```

The `@Deactivate` annotation is described in the above comments. This method is not the same as the `onDeactivate` method of FakeTripwire, as that method is for plug-in deactivation. The `@Deactivate` method represents when the plug-in itself is being stopped or uninstalled (bundle deactivation).

```
    @Modified

    public void modified(Map<String, Object> properties)
    {
        updateConfig(properties);
    }
```

Similar to the `Updated` method described in the FakeTripwire, the modified method controls how the configuration of the bundle is managed. In this case, it simply calls another method to handle the functionality (which will be discussed later). The `@Modified` annotation tells bndtools this is the method to call when the bundle configuration needs updated. In the case of FakeTripwire, the `AssetProxy` abstracts the `@Modified` annotation.

```
    void stop()
    {
        if (m_eventprocessor != null)
        {
            m_eventprocessor.kill = true;
            m_eventprocessor.interrupt();
            try {
                m_eventprocessor.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            m_eventprocessor = null;
        }

        Logging.log(LogService.LOG_INFO,          "SampleConsumer:
STOPPED!!");
    }
```

The `stop` method was user implemented to perform the actions necessary when the plug-in is deactivated.

```
    void updateConfig(Map<String, Object> properties)
    {
        ConsumerConfigInterface consumerconfig =

Configurable.createConfigurable(ConsumerConfigInterface.class,
            properties);
        m_run = consumerconfig.Run();
    }
```

The `updateConfig` method was called from the method bearing the `@Modified` annotation. This method controls how the bundle properties will be updated through use of a configurable just like FakeTripwire.

```
    @Override
    public void handleEvent(Event event) {

        if (!m_run)
        {
            Logging.log(LogService.LOG_INFO,"SampleConsumer::
                  handleEvent...NOT RUNNING....IGNORING EVENT.");
            return;
        }
```

```
        try
        {
            // Check event topic to make sure it is something we
are
            interested in.
            if (event.getTopic().compareTo(
                  ObservationStore.TOPIC_OBSERVATION_PERSISTED) ==
0 ||
                  event.getTopic().compareTo(
                  ObservationStore.TOPIC_OBSERVATION_MERGED) == 0)
            {
                // We are interested in this event. However, we use
a
                // background thread to do the actual
                // processing, because handleEvent() is called on
a framework
                // thread and we need to return
                // as soon as possible. If handleEvent() takes too
long, it
                // can cause the framework to time out
                // and stop sending events to this bundle.

                // Get the UUID for the observation that was just
posted.
                UUID obsUUID = (UUID)event.getProperty(
                ObservationStore.EVENT_PROP_OBSERVATION_UUID);
                Logging.log(LogService.LOG_INFO,"SampleConsumer:
got event
                  UUID: "+ obsUUID.toString());
                // Put the UUID in the queue for processing by the
background
                // thread.
                m_eventQueue.offer(obsUUID);
            }
            else
            {
                // sanity check

Logging.log(LogService.LOG_INFO,"SampleConsumer::handleEvent:
                  unexpected event topic %s", event.getTopic());
                return;
            }
        }
        catch (Exception e)
        {

Logging.log(LogService.LOG_INFO,"SampleConsumer::handleEvent:
                  got exception %s", e.getMessage());
        }
    }
```

The `handleEvent` method is overridden from the EventHandler interface. In this case, the `handleEvent` method will ensure a received event is of a topic that the consumer is associated with, then logs the received event and adds the universally unique identifier (UUID) of the event to the `m_eventQueue` for later use. It is

important that no time-consuming processing be performed in the `handleEvent` method or the operation will time out and be blacklisted. Once blacklisted, the `handleEvent` method will no longer be called. Therefore, it is best to do all processing on another thread.

```
    void init()
    {
        if (!m_run)
        {
            Logging.log(LogService.LOG_INFO,      "SampleConsumer:
disabled by
            configuration");
            return;
        }


        // Start processing thread
        m_eventprocessor = new EventProcThread();
        m_eventprocessor.setName("EventProcessor");
        m_eventprocessor.setDaemon(true);
        m_eventprocessor.start();
    }
```

The `init` method was called from the `activate` method seen earlier. Here we see the consumer is initialized by starting a new event-processing thread through the `EventProcThread` method. This is important because if an operation takes too long without being on a dedicated thread, the operation will time out as mentioned previously. The `EventProcThread` class is shown below.

```
    // Background thread for logging the data
    class EventProcThread extends Thread
    {
        public boolean kill = false;

        @Override
        public void run()
        {

Logging.log(LogService.LOG_INFO,"SampleConsumer::EventProcThread:
                  running");

            while (!kill)
            {
                Observation obs = null;
                try
                {
                    //   Wait   for   an   observation   uuid   from
handleEvent().
                    UUID obsUUID = m_eventQueue.take();
                    // Retrieve the observation from the persistent
store.
                    obs = m_obsStore.find(obsUUID);
```

```
Logging.log(LogService.LOG_INFO,"SampleConsumer::
                        EventProcThread: got Observation from
                        "+obs.getAssetName());
                }
                catch (InterruptedException x)
                {
                    continue;
                }
                catch (Exception x)
                {

Logging.log(LogService.LOG_INFO,"SampleConsumer: error
                        processing    outbound    message:    %s",
x.getMessage());
                    continue;
                }
            }


Logging.log(LogService.LOG_INFO,"SampleConsumer::EventProcThread:
                    STOPPING!!");
        }
    }
```

The final method in this file is `EventProcThread` that extends the built-in Java Thread. This implementation simply gets observations and logs a message stating the observation was received. Because there is no real functionality implemented here, this continues until a kill signal is received.

### 3.2.2 ConsumerConfigInterface.java

The configuration interface of the Consumer is very similar to that of the FakeTripwire. The ConsumerConfigInterface.java file creates an interface as described in the FakeTripwireAssetAttributes.java file.

```
// tells BND this interface provides ConfigurationAdmin data
@OCD(name     =     "Consumer     plug-in")     public     interface
ConsumerConfigInterface {
    // tells BND this is a configuration attribute definition, and
provides a
    // default value
    @AD(required=false, deflt = "true")
    boolean Run();
}
```

### 3.2.3 Summary of the sampleConsumer

The sampleConsumer was developed as an example plug-in to show how an OSUS plug-in could gather observations from the observation store. Although there was not real functionality on the observations after they were gathered, this plug-in demonstrated all of the necessary components to get data from the system. More

annotations were seen such as the `@Modified`, `@Activate`, and `@Deactivate`, which show how bndtools is able to construct the life cycle, attributes, and services in more detail; as in the FakeTripwire asset, the methods with these annotations were abstracted into the implemented `AssetProxy`. Lastly, a technique for handling time-consuming processes was shown, as a separate thread is needed.

## 3.3 Writing a Plug-In

The previous methods have analyzed the functionality of 2 existing OSUS plug-ins: one that consumes data and one that provides it. This section will reiterate many of the topics already discussed, yet frame them in a way that coincides more with development than analysis. By the end of this section (and with a small amount of assistance from the other documents where mentioned), all of the tools needed to write an edge detector plug-in, and run it on an OSUS controller, will have been covered.

### 3.3.1 Edge Detector

The goal of this example is to implement an edge-detection algorithm in OSUS. This tutorial will use concepts from both of the previous walkthroughs, as this plug-in must not only consume data but also provide it. To keep on the simple side, this algorithm implementation will simply grab any image persisted, process it, and then persist (return) the edge profile of the image as a new observation back to the observation store. To test this edge detector, a camera to capture some images is needed. The testing-specific setup will be discussed in the testing section at the end of this section. The full source code can be seen in Appendixes F, G, and H.

The com.acme.sampleConsumer will be used as a starting point, so that project will be copied to a new location and renamed to mil.arl.alg.edgeDetector. Ensure that when renaming this, you update the directory structure to match the name. To make the transition quick, it is easiest to do 2 find-and-replace queries in both of the source files: 1) find “com.acme.sampleConsumer” and replace with “mil.arl.alg.edgeDetector” and 2) find “Consumer” and replace with “edgeDetector”. Another important step is to change the first line in the bnd.bnd file (Private-Package:) to match the package name (e.g., Private-Package: mil.arl.alg.edgeDetector)[*].

Remembering the goal of this plug-in, 3 main steps must be accomplished: 1) receive a persisted image, 2) compute the edge profile of the received image, and 3) persist the edge profile of the image to the observation store as a new observation.

[*] If the starting project is an asset plug-in, there will be a capabilities.xml file that must have its name changed (as mentioned earlier) or the plug-in will not function properly.

The first part of this is very simple. Because we used the sampleConsumer project as a starting point, all of the functionality to receive an image has already been implemented. Therefore, we can move on to Step 2.

In order to get the edge profile of an image, and edge-detection algorithm must be used. This implementation uses the edge-detector class found at http://www.tomgibara.com/computer-vision/canny-edge-detector. This java class was nested into the plug-in for simplicity. Now that we have obtained an edge-detection algorithm, we must pass the image received from the object store to the edge detector. It is very important that all of this be done on a thread or the operation will time out before completion.

The following code snippet shows the event thread that performs the following:

- Takes the UUID of an observation from the event queue
- If the observation has not already been processed, or is not an image that this plug-in generated, proceeds
- Gets an observation from the observation store by UUID
- Gets the digital media from the observation
- Passes the received digital media (image) to the edge detector, and receives back the edge profile as new digital media (more in-depth information on this process later)
- Creates a new observation, and adds the edge-profile digital media and an image metadata object to it
- Populates the observation with the necessary information for persisting (specifics on what is needed for each of the data types can be seen in the XSDs listed in Section 2.4)
- Persists the new observation to the observation store

```
class EventProcThread extends Thread {

    public boolean kill = false;

    @Override
    public void run() {


Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
                running");

        while (!kill) {

            Observation obs = null;
```

```
                try {

                    //  Wait  for  an  observation  uuid  from
handleEvent().
                    UUID obsUUID = m_eventQueue.take();

                    // Retrieve the observation from the persistent
store, if
                    // not already processed.
                    //  This  configuration  is  very  poor,  will  be
updated!
                    if (!processed.contains(obsUUID)) {

                        obs = m_obsStore.find(obsUUID);


Logging.log(LogService.LOG_INFO,"edgeDetector::
                              EventProcThread:  got  Observation
from
                              "+obs.getAssetName());
                        //  to  make  simple,  simply  process  this
observation
                        // and post the processed image
                        DigitalMedia         receivedImg         =
obs.getDigitalMedia();

                        if (receivedImg == null) {

Logging.log(LogService.LOG_INFO,"edgeDetector
                              ::EventProcThread:  Received  Image
was null");
                            continue;
                        }

                        Logging.log(LogService.LOG_INFO,
                              "edgeDetector::EventProcThread:
Processing
                              Received observation");
                        DigitalMedia        processedImg        =
detectEdges(receivedImg);

Logging.log(LogService.LOG_INFO,"edgeDetector
                              ::EventProcThread:          Finished
                              processing,      persisting      new
                              observation");
                        processed.add(obsUUID);

                        // prepare observation for persisting
                        Observation obsImg = new

Observation().withDigitalMedia(processedImg)
                              .withImageMetadata(new
ImageMetadata());
                        UUID newuuid = UUID.randomUUID();
                        obsImg.setUuid(newuuid);
                        processed.add(newuuid);
```

```
obsImg.setSystemInTestMode(_terraHarvestController
                              .getOperationMode()                ==
                             OperationMode.TEST_MODE);
                        obsImg.setVersion(m_obsStore
                              .getObservationVersion());

obsImg.setSystemId(_terraHarvestController.getId());
                        //obsImg.setAssetUuid(serviceuuid);
                        obsImg.setUuid(newuuid);
                        obsImg.setAssetUuid(myassetid);

obsImg.setAssetName("Algorithm:EdgeDetection");
                        obsImg.setAssetType("Algorithm");
                        obsImg.setSensorId(servicepid);
                        obsImg.setCreatedTimestamp(System
                              .currentTimeMillis());
                        // prepare image metadata
                        ImageMetadata imd = new ImageMetadata();
                        imd.setResolution(new   PixelResolution(0,
0));
                        imd.setImageCaptureReason(new

      ImageCaptureReason(ImageCaptureReasonEnum
                              .OTHER, null));
                        imd.setCaptureTime(new
                              Long(System.currentTimeMillis()));
                        imd.setPictureType(PictureTypeEnum
                              .FULL_FIELD_OF_VIEW);
                        imd.setFocus(1.0F);
                        imd.setZoom(1.0F);
                        imd.setColor(true);

imd.setWhiteBalance(WhiteBalanceEnum.AUTO);
                        imd.setChangedPixels(0.0);
                        imd.setImager(new Camera(0, "Alger",
                              CameraTypeEnum.VISIBLE));

                        //try {
                        obsImg.setImageMetadata(imd);
                        m_obsStore.persist(obsImg);
                        //m_Context.persistObservation(obsImg);
                        //} catch (IllegalArgumentException |
                        PersistenceFailedException |
                        ValidationFailedException e) {

//Logging.log(LogService.LOG_ERROR,"edgeDetector:
                            error    persisting    image:    %s",
e.getMessage());
                        //}
                    }
                } catch (InterruptedException x) {

                    continue;

                }
                catch (Exception x) {
```

```
Logging.log(LogService.LOG_ERROR,"edgeDetector: error
                    processing    outbound    message:    %s",
x.getMessage());
                Logging.log(LogService.LOG_ERROR, x,
                    "edgeDetector::EventProcThread: %s", "");
                continue;
            }
        }

Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
            STOPPING!!");
        }
    }
```

As just shown in the code snippet, the newly created object is persisted using the `persist` method of the object-store reference. (If any problems are encountered or any questions arise about the expected composition of an observation, see the XSDs referred to in Section 2.4.) For a better example of how to work with digital media, the following method shows how the `detectEdges` method converts from digital media, to a buffered image, and back:

```
    /**
     * detectEdges computes the edges on the passed in image, and
then
     returns the edge profile of the image.
     *
     * @param dm - the image to process
     * @return - the edge profile of the image
     */
    public DigitalMedia detectEdges(DigitalMedia dm) {

        Logging.log(LogService.LOG_INFO,"edgeDetector:   searching
for
            observation edges");
        // create the detector
        CannyEdgeDetector detector = new CannyEdgeDetector();

        // adjust its parameters as desired
        // this is held for use in future version
        detector.setLowThreshold(m_lowThresh);
        detector.setHighThreshold(m_highThresh);

        // get image from received observation
        byte[] rawimage = dm.getValue();
        InputStream         rawImageStream         =         new
ByteArrayInputStream(rawimage);
        BufferedImage image = null;
        // create buffered image from received image
        try {
            image = ImageIO.read(rawImageStream);
        } catch (IOException e){
```

```
            Logging.log(LogService.LOG_ERROR,"edgeDetector:  error
processing
            input image: %s", e.getMessage());
            return dm;
        }

        //apply detector to received image
        detector.setSourceImage(image);
        detector.process();
        // get resulting edge image
        BufferedImage edges = detector.getEdgesImage();
        // do some converting, image is ARGB, but need RGB for jpg
        BufferedImage img = new BufferedImage(edges.getWidth(),
            edges.getHeight(), BufferedImage.TYPE_INT_RGB);
            img.setRGB(0,  0, edges.getWidth(), edges.getHeight(),
            edges.getRGB(0,          0,          edges.getWidth(),
edges.getHeight(), null, 0,
            edges.getWidth()), 0, edges.getWidth());

        //convert image to byte array for insertion into digital
media object
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] edgeImageBytes = null;
        try {

            ImageIO.write(img, "jpg", baos);
            baos.flush();
            edgeImageBytes = baos.toByteArray();
            baos.close();
            // output image to file for testing purposes
            //        ImageIO.write(img,        "jpg",        new
File(".\\WhatIsHappening.jpg"));

        } catch (IOException e){

            Logging.log(LogService.LOG_ERROR,"edgeDetector:  error
converting
            edge profile image: %s", e.getMessage());
            return dm;

        }

        DigitalMedia  dmEdge  =  new  DigitalMedia(edgeImageBytes,
"image/jpg");

        return dmEdge;
    }
```

As just shown in this code snippet, a digital-image object holds the image data as a byte array (usually in JPEG format), and can be accessed through the `getValue` method of the digital-media object. The byte array can then be converted to a buffered image for use by the edge detector. Converting the buffered image returned from the edge detector back to a byte array (for insertion into a digital-media object) is more difficult, as the chosen edge detector returns an

alpha–red–green–blue (ARGB) image where we want a red–green–blue (RGB) image. After some converting, however, the buffered image is changed to JPEG format and then written to a byte array. Inserting a byte array into a digital-media object is as simple as creating a new digital-media object with arguments for the image byte array and the data type.

The edge detector implemented here takes 2 parameters: a low threshold and a high threshold. As an example of how to implement configurable parameters (or attributes), these were added to the `edgeDetectorConfigInterface`. The edgeDetectorConfigInterface.java is the same as what was previously seen in the FakeTripwire and sampleConsumer asset attributes, yet includes 2 extra values for the threshold parameters:

```
package mil.arl.alg.edgeDetector;

import aQute.bnd.annotation.metatype.Meta.AD;
import aQute.bnd.annotation.metatype.Meta.OCD;

// tells BND this interface provides ConfigurationAdmin data
@OCD(name = "edgeDetector plug-in")
public interface edgeDetectorConfigInterface {
    // tells BND this is a configuration attribute definition, and
provides a
    // default value
    @AD(required=false, deflt = "true")
    boolean Run();
    @AD(required=false, deflt = "0.5")
    float lowThreshold();
    @AD(required=false, deflt = "1")
    float highThreshold();
}
```

This example has 3 defined attributes: a Boolean called Run, a float called lowThreshold, and a float called highThreshold. To reiterate, the configuration process defining these attributes makes them accessible to the OSUS-SG (the GUI, as well as the OSUS controller and other plug-ins) and editable by a user. If any of these attributes are edited, then the edgeDetector will get the updated changes through the method annotated with the `@Modified` tag. In the case of the edgeDetector:

```
    @Modified
    public void modified(Map<String, Object> properties) {

        updateConfig(properties);
        setServicePIDString(properties);

    }
```

The `updateConfig` method is called and the property map sent to it. This property list is completely managed by OSUS and OSGi. By simply defining them in the ConfigInterface file, they are accessible as seen here.

```
    void updateConfig(Map<String, Object> properties) {

        edgeDetectorConfigInterface consumerconfig =

Configurable.createConfigurable(edgeDetectorConfigInterface.class
,
            properties);
        m_run = consumerconfig.Run();
        m_lowThresh = consumerconfig.lowThreshold();
        m_highThresh = consumerconfig.highThreshold();

    }
```

As just seen, a configurable is created for the edgeDetectorConfigInterface, and each of the defined attributes is callable from the created configurable. This allows for access to any newly set attribute; then, they are simply set to a class variable for later use. This makes for very easy attribute modification, management, and creation. All of the "heavy lifting" is done behind the scenes. At this point, the new plug-in can be built for testing.

### 3.3.2 Building and Testing the Plug-in

This section will discuss adding the newly created edge detector plug-in to a controller. Before getting into the compilation and testing of the plug-ins, a camera will be needed to generate some images. The plug-in edu.udayton.udri.asset.webcam.uvc supports any USB video class (UVC)-compliant webcam; therefore, any UVC-compliant webcam can be used for this test.

To compile a plug-in, see Section 5.4 in the *OSUS Plug-in Guide*.[4] Compile the UVC plug-in and add the generated jar file to the "bundles" folder within your controller. Compile the edgeDetector and add the generated jar file to this "bundles" folder as well. Start the OSUS Controller as described in Section 1.3 of this report. Start the OSUS-SG, and connect it to the controller using the instructions found in Section 6.7.2 of the *OSUS-R Operator Instructions*.[6]

Follow the instructions found in Section 6.9.1 in *OSUS-R Operator Instructions*[6] to ensure both the UVC plug-in and the edgeDetector plug-in are installed and active on your controller. Then, follow the instructions found in Section 6.10.2 of those instructions[6] to add the camera asset to the controller. Now, activate the camera asset.

The camera is now active and ready to capture data, and the edgeDetector is ready to process observations. Select the Capture Data button as described in Section 6.10.5.7 of the operator instructions[6] and then go to the observations tab as described in Section 6.10.6. At this point you should see (at least) 2 observations: one image from the camera and the edge profile as an image from the edge detector. For troubleshooting, ensure the edgeDetector code is correct and refresh the OSUS-SG webpage. If this is not the problem, see Sections 5.7, 5.8, and 5.9 of the plug-in guide[4] for more detail on troubleshooting and plug-in testing.

### 3.3.3 Summary of the EdgeDetector Plug-in

This plug-in used the sampleConsumer plug-in as a starting point. Functionality was added that allows for the processing of received images and then the persisting of the results back to the observation store. This example shows the proper use of the @activate, @modified, and @deactivate annotations as well as how to process information without blacklisting the plug-in due to timely computations. Lastly, this example presented how to persist an observation to the observation store using a reference received by the OSGi runtime during initialization.

## 4. Summary and Conclusion

OSUS is a sophisticated standard that decreases development time, decreases code complexity, and increases component interoperability. Furthermore, placing OSUS within the main embedded PC or microcontroller of a sensor system allows the system to leverage all of the existing OSUS capabilities as well as take advantage of the OSUS bridges, interfaces, and mission programs that have already been developed.

This guide provides the basics of OSUS and its underlying framework, OSGi, while referring to other documentation for greater detail. (Appendix I, additionally, has an OSUS plug-in compliance checklist.) Moreover, practical application of these basic principles is shown through the analysis of 2 existing plug-ins and the simulated development of a third plug-in. Although the OSUS learning curve can be steep, this report aims to make familiarization much simpler and reduce the time needed to achieve proficiency in plug-in development.

## 5. References


1. OSUS–open standards for unattended sensors. Adelphi (MD): Army Research Laboratory (US); 2016 Sep 13 [accessed 2017 Dec 11]. https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/OSUS.Standard.1.1.0.pdf.

2. Fauth D. Getting started with OSGi declarative services. Vogella Blog. [accessed 2017 Oct 12]. http://blog.vogella.com/2016/06/21/getting-started-with-osgi-declarative-services/.

3. OSGi Alliance. The dynamic module system for java. San Ramon (CA): c2017 [accessed 2017 Oct 12]. https://www.osgi.org/.

4. OSUS plug-in guide. Adelphi (MD): Army Research Laboratory (US); 2016 Oct 28 [accessed 2017 Dec 11]. https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/OSUS.Plug-in.Guide.1.1.0.pdf.

5. OSUS mission programming guide. Adelphi (MD): Army Research Laboratory (US); 2016 Sep 13 [accessed 2017 Dec 11]. https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/OSUS.Mission.Programming.Guide.1.1.0.pdf.

6. OSUS-R operator instructions. Adelphi (MD): Army Research Laboratory (US); 2016 Sep 13 [accessed 2017 Dec 11]. https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/OSUS-R.Operator.Instructions.1.1.0.pdf.

7. OSUS remote interface specification. Adelphi (MD): Army Research Laboratory (US); 2016 Sep 13 [accessed 2017 Dec 11]. https://github.com/ssg-udri/OSUS-R/releases/download/1.1.0/OSUS.Remote.Interface.Specification.1.1.0.pdf.

8. Apache Felix. OSGi framework and service platform. [accessed 2017 Oct 12]. http://felix.apache.org.

9. Thangarajah K. Kishanthan's blog: Using annotations with osgi declarative services. [accessed 2017 Mar 29]. https://kishanthan.wordpress.com/2014/03/29/using-annotation-with-osgi-declarative-services/.

10. TechTarget SearchNetworking. Definition: OSGi (open service gateway initiative). Newton (MA): c2000–2017 [accessed 2017 Mar 16]. http://searchnetworking.techtarget.com/definition/OSGi.

# Appendix A. simpleFakeTripwireAsset.java

This appendix appears in its original form, without editorial change.

```java
package com.acme.assets.simpleFakeTripwire;

import java.util.Map;
import java.util.Random;
import java.util.Set;
import java.util.Timer;
import java.util.TimerTask;

import aQute.bnd.annotation.component.Component;
import aQute.bnd.annotation.metatype.Configurable;
import mil.dod.th.core.asset.Asset;
import mil.dod.th.core.asset.AssetContext;
import mil.dod.th.core.asset.AssetException;
import mil.dod.th.core.asset.AssetProxy;
import mil.dod.th.core.asset.commands.Command;
import mil.dod.th.core.asset.commands.Response;
import mil.dod.th.core.commands.CommandExecutionException;
import mil.dod.th.core.types.FrequencyKhz;
import mil.dod.th.core.types.SensingModality;
import mil.dod.th.core.types.SensingModalityEnum;
import mil.dod.th.core.types.detection.DetectionTypeEnum;
import mil.dod.th.core.types.detection.TargetClassificationType;
import
mil.dod.th.core.types.detection.TargetClassificationTypeEnum;
import mil.dod.th.core.types.status.OperatingStatus;
import mil.dod.th.core.types.status.SummaryStatusEnum;
import mil.dod.th.core.factory.Extension;
import mil.dod.th.core.factory.FactoryException;
import mil.dod.th.core.log.Logging;
import mil.dod.th.core.observation.types.Detection;
import mil.dod.th.core.observation.types.Observation;
import mil.dod.th.core.observation.types.Status;
import mil.dod.th.core.observation.types.TargetClassification;

import org.osgi.service.log.LogService;

/**
 * Fake tripwire, for testing and training purposes.
 *
 * @author jkovach, jtyo
 */
@Component(factory = Asset.FACTORY)
public class simpleFakeTripwireAsset implements AssetProxy {

    // property variables
    int interval;
    int statusInterval;
    SensingModalityEnum modality;

    // class variable for housing data generator
    DataGenerator dg = null;

    /**
     * Class constructor.
     */
    public simpleFakeTripwireAsset() {
```

```
        // super in this case calls java's object class
constructor, which does nothing
        super();

        // create a new DataGenerator
        dg = new DataGenerator();
    }

    /**
     * Reference to the context which provides this class with
methods to
     * interact with the rest of the system.
     */
    private AssetContext m_Context;

    /**
     * initialize is called when all of the plugin's dependencies
are met, and it can be created
     */
    @Override
    public void initialize(final AssetContext context, final
Map<String, Object> props) throws FactoryException {
        // set the provided context to the class variable for
later use
        m_Context = context;
        // set the initial properties of this plugin
        updated(props);

        // Set the plug-in's status message
        m_Context.setStatus(SummaryStatusEnum.OFF, "Asset
Inactive");
    }

    /**
     * updated is called when the properties of the plugin are
changed and need updated
     */
    @Override
    public void updated(final Map<String, Object> props) {
        // create a configurable from the input properties.  This
is used to set the user defined or modified properties to class
variables
        final simpleFakeTripwireAssetAttributes config =
Configurable.createConfigurable(simpleFakeTripwireAssetAttributes
.class,
                props);

        //set the class variables to the input properties
        interval = config.interval();
        statusInterval = config.statusInterval();
        modality = config.modality();

        // restart the data generation functionality
        dg.startOrStopTimer();
    }

    /**
```

```
     * onActivate is called when the plugin is to be activated
     */
    @Override
    public void onActivate() throws AssetException {
        // When the asset is activated, we will generate a random
detection at
        // periodic intervals
        // (delay is specified in the asset configuration) and
post it to the
        // persistent store.

        // set the data generator to active
        dg.active = true;
        // Start the timer that will generate the periodic image.
        dg.startOrStopTimer();

        // Log an activation method
        Logging.log(LogService.LOG_INFO, "Fake Tripwire
activated");

        // this will also generate/send a status observation
        m_Context.setStatus(SummaryStatusEnum.GOOD, "Asset
Activated");
    }

    /**
     * onDeactivate is called when the asset is deactivated, and
therefore must
     */
    @Override
    public void onDeactivate() throws AssetException {
        // Log a deactivation message
        Logging.log(LogService.LOG_INFO, "Fake Tripwire
deactivated");

        // Set variable to stop the timer
        dg.active = false;
        // Stop the timer.
        dg.startOrStopTimer();

        // Set the asset status accordingly
        m_Context.setStatus(SummaryStatusEnum.OFF, "Asset
Deactivated");
    }


    /**
     * onCaptureData is called when the asset is to capture data.
This method will call the
     * observation creation/capturing methods, and then return
the new observation.
     */
    @Override
    public Observation onCaptureData() {
        // Generate and return a single observation. (The base
class will handle
        // posting the observation
```

```
        // to the persistent store.)
        Observation obs = dg.generateObservation(null);

        // Log a message accordingly
        Logging.log(LogService.LOG_INFO, "Fake Tripwire data
captured");

        // Return the generated observation
        return obs;
    }


    /**
     * onPerformBit performs some type of self checking, and then
returns a status with respect to
     * the health of the plugin.
     */
    @Override
    public Status onPerformBit() {
        // onPerformBit should be a health checking measure,
however as this plugin is a data generator
        // there is nothing to check
        // Log an appropriate message
        Logging.log(LogService.LOG_INFO, "Performing Fake
Tripwire BIT");
        // return a status representing the results of the self
check
        return new Status().withSummaryStatus(new
OperatingStatus(SummaryStatusEnum.GOOD, "BIT Passed"));
    }

    /**
     *
     */
    @Override
    public Response onExecuteCommand(final Command
capabilityCommand) throws CommandExecutionException {
        // No commands currently supported.
        throw new CommandExecutionException("This asset does not
support any commands.");
    }

    /**
     *
     */
    @Override
    public Set<Extension<?>> getExtensions() {
        // Currently not implemented
        return null;
    }


    /**
     * Class to handle data generation
     *
     * This class handles all of the data generation of the plug-
in and is not osus specific.
```

```
     * This class is out of scope of this discussion and will not
be discussed.
     */
    private class DataGenerator {
        // Background task that periodically generates an image
and posts it to
        // the store.
        Random prng;
        public boolean active = false;

        Timer theObservationTimer = null;
        Timer theStatusTimer = null;

        String[] targetIdList = { "11111", "22222", "33333",
"44444", "55555", "66666", "77777", "88888", "99999", "AAAAA",
                "BBBBB", "CCCCC", "DDDDD", "EEEEE", "FFFFF",
"GGGGG", "HHHHH", "IIIII", "JJJJJ", "KKKKK" };

        public DataGenerator() {
            prng = new Random();
        }

        protected class ObservationMaker extends TimerTask {
            @Override
            public void run() {
                generateAndPostDetection(null);
            }
        }

        protected class StatusMaker extends TimerTask {
            @Override
            public void run() {
                generateAndPostStatus();
            }
        }

        protected void startOrStopTimer() {
            if (theObservationTimer != null) {
                theObservationTimer.cancel();
                theObservationTimer = null;
            }
            if (theStatusTimer != null) {
                theStatusTimer.cancel();
                theStatusTimer = null;
            }

            if (active) {
                if (interval > 0) {
                    theObservationTimer = new Timer(true);
                    theObservationTimer.schedule(new
ObservationMaker(), System.currentTimeMillis() % interval,
                            interval);
                }
                if (statusInterval > 0) {
                    theStatusTimer = new Timer(true);
                    theStatusTimer.schedule(new StatusMaker(),
statusInterval, statusInterval);
```

```
                }
            }
        }

        protected void generateAndPostStatus() {
            try {
                // Generate an observation.
                Observation o = generateStatus();

                // Post it to the persistent store.
                // postObservation is a base class method that
does this for us.
                m_Context.persistObservation(o);
                Logging.log(LogService.LOG_INFO, "fake tripwire
posted a status message");
            } catch (Exception e) {
                Logging.log(LogService.LOG_ERROR, e, "fake
tripwire %s could NOT post status!", m_Context.getName());
            }
        }

        protected void generateAndPostDetection(String targetId)
{
            try {

                // Generate an observation.
                Observation o = generateObservation(targetId);

                // Post it to the persistent store.
                // postObservation is a base class method that
does this for us.

                m_Context.persistObservation(o);
                Logging.log(LogService.LOG_INFO, "fake tripwire
%s posted a detection, modality = %s",
                        m_Context.getName(), modality);
            } catch (Exception e) {
                Logging.log(LogService.LOG_ERROR, e, "fake
tripwire %s could NOT post detection! %s",
                        m_Context.getName());
            }
        }

        protected Observation generateStatus() {
            Observation obs = new Observation();

            // system will take care of setting the location,
asset id, etc in
            // the observation
            // after we return it

            // obs.setSensorId("0");

            // Send some status too, while we're at it
            Status s = new Status();
            s.setSummaryStatus(new
OperatingStatus().withSummary(SummaryStatusEnum.GOOD));
```

```
            s.setSensorFov(10.0);
            obs.setStatus(s);

            return obs;
        }

        // Generates observation data.
        protected Observation generateObservation(String
targetId) {
            Observation obs = new Observation();

            // obs.setSensorId("0");
            obs.getModalities().add(new SensingModality(modality,
""));

            // add some detection data
            Detection dd = new Detection();
            dd.setType(DetectionTypeEnum.ALARM);
            dd.getTargetClassifications()
                    .add(new TargetClassification()
                            .withType(new
TargetClassificationType().withValue(TargetClassificationTypeEnum

.values()[prng.nextInt(TargetClassificationTypeEnum.values().leng
th)])));

            if (targetId != null) {
                dd.setTargetId(targetId);
            } else if (modality == SensingModalityEnum.RFID) {

dd.setTargetId(targetIdList[prng.nextInt(targetIdList.length)]);
            } else if (modality == SensingModalityEnum.RADIATION)
{
                dd.setTargetFrequency(new
FrequencyKhz().withValue(prng.nextDouble() * 1000));
            }

            obs.setDetection(dd);

            return obs;
        }

    }

}
```

## Appendix B. simpleFakeTripwireAssetAttributes.java

This appendix appears in its original form, without editorial change.

```java
package com.acme.assets.simpleFakeTripwire;

import aQute.bnd.annotation.metatype.Meta.AD;
import aQute.bnd.annotation.metatype.Meta.OCD;
import mil.dod.th.core.ConfigurationConstants;
import mil.dod.th.core.asset.AssetAttributes;
import mil.dod.th.core.types.SensingModalityEnum;

/**
 * Interface which defines the configurable properties for a
simpleFakeTripwire.
 */
@OCD(description =
ConfigurationConstants.PARTIAL_OBJECT_CLASS_DEFINITION)
public interface simpleFakeTripwireAssetAttributes extends
AssetAttributes {
    /**
     * Each method annotated with @AD becomes a configuration
property available
     * to the plug-in. The return type of each method is the type
of the
     * configuration property. All simple types are supported and
other types
     * are supported if they can be converted from a string
(e.g., class with a
     * constructor accepting a string or an enum). Also, the type
can be an
     * array or a collection for properties with multiple values.
     */

    @AD(required = false, deflt = "60000", name = "Detection
Interval", description = "Detection Generation Interval
(milliseconds)")
    int interval();

    @AD(required = false, deflt = "300000", name = "Status
Interval", description = "Status Generation Interval
(milliseconds)")
    int statusInterval();

    @AD(required = false, deflt = "PIR", name = "Modality",
description = "Modaility (from SensingModalityEnum)")
    SensingModalityEnum modality();

}
```

# Appendix C. simpleFakeTripwire capabilities-xml com.acme.assets.simpleFakeTripwire.simpleFakeTripwireAsset.xml

This appendix appears in its original form, without editorial change.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:AssetCapabilities xmlns="capability.core.th.dod.mil"
xmlns:ns2="capability.asset.core.th.dod.mil"
xmlns:ns3="capability.link.ccom.core.th.dod.mil"
xmlns:ns4="capability.transport.ccom.core.th.dod.mil"
xmlns:ns5="capability.physical.ccom.core.th.dod.mil"
manufacturer="Acme Corporation, Tactical Systems Division"
description="Generates random detections of various types"
productName="Acme Fake Tripwire">
    <primaryImage encoding="image/jpeg">/9j</primaryImage>
    <secondaryImages encoding="image/jpeg">/9j</secondaryImages>
    <ns2:modalities description="none" value="Imager"/>
    <ns2:minRange>0.0</ns2:minRange>
    <ns2:maxRange>20.0</ns2:maxRange>
    <ns2:nominalRange>20.0</ns2:nominalRange>
    <ns2:minFov>20.0</ns2:minFov>
    <ns2:maxFov>20.0</ns2:maxFov>
    <ns2:nominalFov>20.0</ns2:nominalFov>
    <ns2:detectionCapabilities targetId="false"
directionOfTravel="false" trackHistory="false"
targetFrequency="false" targetLOB="false"
targetOrientation="false" targetRange="false" targetSpeed="false"
targetLocation="false">
        <ns2:typesAvailable>Alarm</ns2:typesAvailable>
        <ns2:typesAvailable>Test</ns2:typesAvailable>
        <ns2:classifications value="Other" />
    </ns2:detectionCapabilities>
    <ns2:commandCapabilities performBIT="true"
captureData="true">

<ns2:supportedCommands>GetPositionCommand</ns2:supportedCommands>

<ns2:supportedCommands>GetVersionCommand</ns2:supportedCommands>

<ns2:supportedCommands>SetPositionCommand</ns2:supportedCommands>
    </ns2:commandCapabilities>
</ns2:AssetCapabilities>
```

*The primaryImage and secondaryImages lines contain the base64 encoded image. For readability, the contents of those tags was shortened to “/9j”.

## Appendix D. simpleFakeTripwireAssetScanner.java

This appendix appears in its original form, without editorial change.

```java
package com.acme.assets.simpleFakeTripwire;

import java.util.HashMap;
import java.util.Set;

import aQute.bnd.annotation.component.Component;
import mil.dod.th.core.asset.Asset;
import mil.dod.th.core.asset.AssetDirectoryService.ScanResults;
import mil.dod.th.core.asset.AssetException;
import mil.dod.th.core.asset.AssetScanner;
import mil.dod.th.core.factory.ProductType;

/**
 * TODO: Optional class. Implement the scanForNewAssets method or
remove this
 * class if scanning is not supported.
 *
 * Example SDK Plug-in scanner implementation.
 *
 * @author jkovach
 */
@Component
@ProductType(simpleFakeTripwireAsset.class)
public class simpleFakeTripwireAssetScanner implements
AssetScanner {
    @Override
    public void scanForNewAssets(final ScanResults results, final
Set<Asset> existing) throws AssetException {
        boolean alreadyHave = false;

        for (Asset a : existing) {
            if
(a.getFactory().getProductType().equals(simpleFakeTripwireAsset.c
lass.getName())) {
                alreadyHave = true;
                break;
            }
        }

        if (!alreadyHave) {
            results.found("simpleFakeTripwire1", new
HashMap<String, Object>());
        }
    }
}
```

# Appendix E. Consumer.java

This appendix appears in its original form, without editorial change.

```
package com.acme.sampleConsumer;

import java.util.Map;
import java.util.UUID;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import mil.dod.th.core.log.Logging;
import mil.dod.th.core.observation.types.Observation;
import mil.dod.th.core.persistence.ObservationStore;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventConstants;
import org.osgi.service.event.EventHandler;
import org.osgi.service.log.LogService;

import aQute.bnd.annotation.component.*;
import aQute.bnd.annotation.metatype.Configurable;

@Component(provide={EventHandler.class}, /* provides EventHandler
service to receive OSGi events */
    immediate=true, /* activate always even if no consumers */
    designate=ConsumerConfigInterface.class, /* class containing
config info for the metatype and config admin services */
    configurationPolicy=ConfigurationPolicy.optional, /* activate
bundle even if configuration does not exist */
    properties={EventConstants.EVENT_TOPIC + "=" +
ObservationStore.TOPIC_OBSERVATION_PERSISTED
     + "|" + ObservationStore.TOPIC_OBSERVATION_MERGED, /*
register for events on this topic */
     } )
public class Consumer implements EventHandler {

    Boolean m_run = false;

    //  queue holding UUIDs to be processed by the logger thread
    BlockingQueue<UUID> m_eventQueue = new
LinkedBlockingQueue<UUID>();
    EventProcThread m_eventprocessor = null;

    ObservationStore m_obsStore = null;
    @Reference
    // get reference to the ObservationStore service so we can
retrieve observations after
    // they are posted
    // This method is called by the framework due to the
@Reference annotation.
    public void setObservationStore(ObservationStore obsStore)
    {
        m_obsStore = obsStore;
    }

    @Activate // <- tells bnd this is the activate method
    // activate method called by the framework when all
dependences have been satisfied and
    // the bundle should start processing
    public void activate(Map<String, Object> properties)
```

```
    {
        updateConfig(properties);
        init();
    }
    @Deactivate // <- tells bnd this is the deactivate method
    // deactivate method called by the framework when the bundle
should be shut down
    // because the framework is stopping/the bundle is being
uninstalled/etc.
    public void deactivate()
    {
        stop();
    }

    @Modified
    public void modified(Map<String, Object> properties)
    {
        updateConfig(properties);
    }

    void stop()
    {
        if (m_eventprocessor != null)
        {
            m_eventprocessor.kill = true;
            m_eventprocessor.interrupt();
            try {
                m_eventprocessor.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            m_eventprocessor = null;
        }

        Logging.log(LogService.LOG_INFO, "SampleConsumer:
STOPPED!!");
    }
    void updateConfig(Map<String, Object> properties)
    {
        ConsumerConfigInterface consumerconfig =
Configurable.createConfigurable(ConsumerConfigInterface.class,pro
perties);
        m_run = consumerconfig.Run();
    }

    @Override
    public void handleEvent(Event event) {

        if (!m_run)
        {

Logging.log(LogService.LOG_INFO,"SampleConsumer::handleEvent...NO
T RUNNING....IGNORING EVENT.");
            return;
        }

        try
```

```
        {
            // Check event topic to make sure it is something we
are interested in.
            if
(event.getTopic().compareTo(ObservationStore.TOPIC_OBSERVATION_PE
RSISTED) == 0
                    ||
event.getTopic().compareTo(ObservationStore.TOPIC_OBSERVATION_MER
GED) == 0)
            {
                // We are interested in this event. However, we
use a background thread to do the actual
                // processing, because handleEvent() is called on
a framework thread and we need to return
                // as soon as possible. If handleEvent() takes
too long, it can cause the framework to time out
                // and stop sending events to this bundle.

                // Get the UUID for the observation that was just
posted.
                UUID obsUUID =
(UUID)event.getProperty(ObservationStore.EVENT_PROP_OBSERVATION_U
UID);
                Logging.log(LogService.LOG_INFO,"SampleConsumer:
got event UUID: "+ obsUUID.toString());
                // Put the UUID in the queue for processing by
the background thread.
                m_eventQueue.offer(obsUUID);
            }
            else
            {
                // sanity check

Logging.log(LogService.LOG_INFO,"SampleConsumer::handleEvent:
unexpected event topic %s", event.getTopic());
                return;
            }
        }
        catch (Exception e)
        {

Logging.log(LogService.LOG_INFO,"SampleConsumer::handleEvent: got
exception %s", e.getMessage());
        }

    }
    void init()
    {
        if (!m_run)
        {
            Logging.log(LogService.LOG_INFO, "SampleConsumer:
disabled by configuration");
            return;
        }


        // Start processing thread
```

```
        m_eventprocessor = new EventProcThread();
        m_eventprocessor.setName("EventProcessor");
        m_eventprocessor.setDaemon(true);
        m_eventprocessor.start();
    }

    // Background thread for logging the data
    class EventProcThread extends Thread
    {
        public boolean kill = false;

        @Override
        public void run()
        {

Logging.log(LogService.LOG_INFO,"SampleConsumer::EventProcThread:
running");

            while (!kill)
            {
                Observation obs = null;
                try
                {
                    // Wait for an observation uuid from
handleEvent().
                    UUID obsUUID = m_eventQueue.take();
                    // Retrieve the observation from the
persistent store.
                    obs = m_obsStore.find(obsUUID);

Logging.log(LogService.LOG_INFO,"SampleConsumer::EventProcThread:
got Observation from "+obs.getAssetName());
                }
                catch (InterruptedException x)
                {
                    continue;
                }
                catch (Exception x)
                {

Logging.log(LogService.LOG_INFO,"SampleConsumer: error processing
outbound message: %s", x.getMessage());
                    continue;
                }
            }


Logging.log(LogService.LOG_INFO,"SampleConsumer::EventProcThread:
STOPPING!!");
        }
    }

}
```

INTENTIONALLY LEFT BLANK.

# Appendix F. ConsumerConfigInterface.java

This appendix appears in its original form, without editorial change.

```
package com.acme.sampleConsumer;

import aQute.bnd.annotation.metatype.Meta.AD;
import aQute.bnd.annotation.metatype.Meta.OCD;

@OCD(name = "Consumer plug-in") // <- tells BND this interface
provides ConfigurationAdmin data
public interface ConsumerConfigInterface {
    @AD(required=false, deflt = "true") // <- tells BND this is a
configuration attribute definition, and provides a default value
    boolean Run();
}
```

# Appendix G. edgeDetector.java

This appendix appears in its original form, without editorial change.

```
package mil.arl.alg.edgeDetector;

import java.util.Map;
import java.util.UUID;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import mil.dod.th.core.controller.TerraHarvestController;
import
mil.dod.th.core.controller.TerraHarvestController.OperationMode;
import mil.dod.th.core.log.Logging;
import mil.dod.th.core.observation.types.ImageMetadata;
import mil.dod.th.core.observation.types.Observation;
import mil.dod.th.core.persistence.ObservationStore;
import mil.dod.th.core.types.DigitalMedia;
import mil.dod.th.core.types.image.Camera;
import mil.dod.th.core.types.image.CameraTypeEnum;
import mil.dod.th.core.types.image.ImageCaptureReason;
import mil.dod.th.core.types.image.ImageCaptureReasonEnum;
import mil.dod.th.core.types.image.PictureTypeEnum;
import mil.dod.th.core.types.image.PixelResolution;
import mil.dod.th.core.types.image.WhiteBalanceEnum;

import org.osgi.framework.BundleContext;
import org.osgi.service.event.Event;
import org.osgi.service.event.EventConstants;
import org.osgi.service.event.EventHandler;
import org.osgi.service.log.LogService;

import aQute.bnd.annotation.component.*;
import aQute.bnd.annotation.metatype.Configurable;
import mil.arl.alg.edgeDetector.edgeDetectorConfigInterface;

// imports for the edge detection
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.io.InputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;



@Component(provide={EventHandler.class}, /* provides EventHandler
service to receive OSGi events */
    immediate=true, /* activate always even if no consumers */
    designate=edgeDetectorConfigInterface.class, /* class
containing config info for the metatype and config admin services
*/
    configurationPolicy=ConfigurationPolicy.optional, /* activate
bundle even if configuration does not exist */
    // this property filtering should be okay, as the photos
posted under concerned are persisted
```

```
    properties={EventConstants.EVENT_TOPIC + "=" +
ObservationStore.TOPIC_OBSERVATION_PERSISTED
     + "|" + ObservationStore.TOPIC_OBSERVATION_MERGED, /*
register for events on this topic */
     } )
public class edgeDetector implements EventHandler {

    //Config Variables
    Boolean m_run = false;
    Float m_lowThresh = null;
    Float m_highThresh = null;

    //  queue holding UUIDs to be processed by the logger thread
    BlockingQueue<UUID> m_eventQueue = new
LinkedBlockingQueue<UUID>();
    EventProcThread m_eventprocessor = null;
    List<UUID> processed = new ArrayList<UUID>();
    TerraHarvestController _terraHarvestController = null;
    String servicepid = "unknown";
    String serviceuuidstring = "unknown";
    UUID serviceuuid;
    UUID myassetid = UUID.randomUUID(); // This will probably
need set at some point - 17 Jan 2017
    ObservationStore m_obsStore = null;



    // get reference to the ObservationStore service so we can
retrieve observations after
    // they are posted
    // This method is called by the framework due to the
@Reference annotation.
    @Reference
    public void setObservationStore(ObservationStore obsStore) {

        m_obsStore = obsStore;

    }


    // tells bnd this is the activate method
    // activate method called by the framework when all
dependences have been satisfied and
    // the bundle should start processing
    @Activate
    public void activate(Map<String, Object> properties) {

        updateConfig(properties);
        init();
        setServicePIDString(properties);

    }


    // tells bnd this is the deactivate method
    // deactivate method called by the framework when the bundle
should be shut down
```

```
    // because the framework is stopping/the bundle is being
uninstalled/etc.
    @Deactivate
    public void deactivate() {

        stop();

    }


    // get reference to the setEventAdmin service used to post
events
    @Reference
    public void setTerraHarvestController(TerraHarvestController
ths) {

        _terraHarvestController = ths;

    }


    @Modified
    public void modified(Map<String, Object> properties) {

        updateConfig(properties);
        setServicePIDString(properties);

    }


    private void setServicePIDString(Map<String, Object>
properties) {

        try {

            String temp = ((String)
properties.get("service.pid"));
            serviceuuidstring =
temp.substring(temp.lastIndexOf(".")+1);

        } catch (NullPointerException e) {

            Logging.log(LogService.LOG_INFO,"edgeDetector: error
setting service PID, creating random UUID: %s", e.getMessage());
            serviceuuidstring = UUID.randomUUID().toString();

        }

        serviceuuid = UUID.fromString(serviceuuidstring);
        servicepid = "edgeDetector." + serviceuuidstring;

    }


    void stop() {

        if (m_eventprocessor != null) {
```

```
            m_eventprocessor.kill = true;
            m_eventprocessor.interrupt();

            try {
                m_eventprocessor.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            m_eventprocessor = null;
        }

        Logging.log(LogService.LOG_INFO, "edgeDetector:
STOPPED!!");
    }


    void updateConfig(Map<String, Object> properties) {

        edgeDetectorConfigInterface consumerconfig =
Configurable.createConfigurable(edgeDetectorConfigInterface.class
, properties);
        m_run = consumerconfig.Run();
        m_lowThresh = consumerconfig.lowThreshold();
        m_highThresh = consumerconfig.highThreshold();

    }


    @Override
    public void handleEvent(Event event) {

        if (!m_run) {


Logging.log(LogService.LOG_INFO,"edgeDetector::handleEvent...NOT
RUNNING....IGNORING EVENT.");
            return;

        }

        try {

            // Check event topic to make sure it is something we
are interested in.
            if
(event.getTopic().compareTo(ObservationStore.TOPIC_OBSERVATION_PE
RSISTED) == 0
                    ||
event.getTopic().compareTo(ObservationStore.TOPIC_OBSERVATION_MER
GED) == 0) {
                // We are interested in this event. However, we
use a background thread to do the actual
                // processing, because handleEvent() is called on
a framework thread and we need to return
```

```
                // as soon as possible. If handleEvent() takes
too long, it can cause the framework to time out
                // and stop sending events to this bundle.

                // Get the UUID for the observation that was just
posted.
                UUID obsUUID =
(UUID)event.getProperty(ObservationStore.EVENT_PROP_OBSERVATION_U
UID);
                Logging.log(LogService.LOG_INFO,"edgeDetector:
got event UUID: "+ obsUUID.toString());
                // Put the UUID in the queue for processing by
the background thread.
                m_eventQueue.offer(obsUUID);

            } else {

                // sanity check

Logging.log(LogService.LOG_INFO,"edgeDetector::handleEvent:
unexpected event topic %s", event.getTopic());
                return;

            }
        } catch (Exception e) {


Logging.log(LogService.LOG_INFO,"edgeDetector::handleEvent: got
exception %s", e.getMessage());

        }
    }


    void init() {

        Logging.log(LogService.LOG_INFO, "edgeDetector:
Initializing");

        if (!m_run) {

            Logging.log(LogService.LOG_INFO, "edgeDetector:
disabled by configuration");
            return;

        }

        // Start processing thread
        Logging.log(LogService.LOG_INFO, "edgeDetector: Start
processing thread");
        m_eventprocessor = new EventProcThread();
        m_eventprocessor.setName("EventProcessor");
        m_eventprocessor.setDaemon(true);
        m_eventprocessor.start();

    }
```

```
    // Background thread for logging the data
    class EventProcThread extends Thread {

        public boolean kill = false;

        @Override
        public void run() {


Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
running");

            while (!kill) {

                Observation obs = null;
                try {

                    // Wait for an observation uuid from
handleEvent().
                    UUID obsUUID = m_eventQueue.take();

                    // Retrieve the observation from the
persistent store, if not already processed.
                    // This configuration is very poor, will be
updated!
                    if (!processed.contains(obsUUID)) {

                        obs = m_obsStore.find(obsUUID);

Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
got Observation from "+obs.getAssetName());
                        // to make simple, simply process this
observation and post the processed image
                        DigitalMedia receivedImg =
obs.getDigitalMedia();

                        if (receivedImg == null) {

Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
Received Image was null");
                            continue;
                        }


Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
Processing received observation");
                        DigitalMedia processedImg =
detectEdges(receivedImg);

Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
Finished processing, persisting new observation");
                        processed.add(obsUUID);

                        // prepare observation for persisting
```

```
                        Observation obsImg = new
Observation().withDigitalMedia(processedImg).withImageMetadata(ne
w ImageMetadata());
                        UUID newuuid = UUID.randomUUID();
                        obsImg.setUuid(newuuid);
                        processed.add(newuuid);

obsImg.setSystemInTestMode(_terraHarvestController.getOperationMo
de() == OperationMode.TEST_MODE);

obsImg.setVersion(m_obsStore.getObservationVersion());

obsImg.setSystemId(_terraHarvestController.getId());
                        //obsImg.setAssetUuid(serviceuuid);
                        obsImg.setAssetUuid(myassetid);

obsImg.setAssetName("Algorithm:EdgeDetection");
                        obsImg.setAssetType("Algorithm");
                        obsImg.setSensorId(servicepid);

obsImg.setCreatedTimestamp(System.currentTimeMillis());
                        // prepare image metadata
                        ImageMetadata imd = new ImageMetadata();
                        imd.setResolution(new PixelResolution(0,
0));
                        imd.setImageCaptureReason(new
ImageCaptureReason(ImageCaptureReasonEnum.OTHER, null));
                        imd.setCaptureTime(new
Long(System.currentTimeMillis()));

imd.setPictureType(PictureTypeEnum.FULL_FIELD_OF_VIEW);
                        imd.setFocus(1.0F);
                        imd.setZoom(1.0F);
                        imd.setColor(true);

imd.setWhiteBalance(WhiteBalanceEnum.AUTO);
                        imd.setChangedPixels(0.0);
                        imd.setImager(new Camera(0, "Alger",
CameraTypeEnum.VISIBLE));

                        //try {
                        obsImg.setImageMetadata(imd);
                        m_obsStore.persist(obsImg);
                        //m_Context.persistObservation(obsImg);
                        //} catch (IllegalArgumentException |
PersistenceFailedException | ValidationFailedException e) {

//Logging.log(LogService.LOG_ERROR,"edgeDetector: error
persisting image: %s", e.getMessage());
                        //}
                    }
                } catch (InterruptedException x) {

                    continue;

                }
                catch (Exception x) {
```

```
Logging.log(LogService.LOG_ERROR,"edgeDetector: error processing
outbound message: %s", x.getMessage());
                    Logging.log(LogService.LOG_ERROR, x,
"edgeDetector::EventProcThread: %s", "");
                    continue;
                }
            }


Logging.log(LogService.LOG_INFO,"edgeDetector::EventProcThread:
STOPPING!!");
        }
    }

    /**
     * detectEdges computes the edges on the passed in image, and
then returns the edge profile of the image.
     *
     * @param dm - the image to process
     * @return - the edge profile of the image
     */
    public DigitalMedia detectEdges(DigitalMedia dm) {

        Logging.log(LogService.LOG_INFO,"edgeDetector: searching
for observation edges");
        // create the detector
        CannyEdgeDetector detector = new CannyEdgeDetector();

        // adjust its parameters as desired
        // this is held for use in future version
        detector.setLowThreshold(m_lowThresh);
        detector.setHighThreshold(m_highThresh);

        // get image from received observation
        byte[] rawimage = dm.getValue();
        InputStream rawImageStream = new
ByteArrayInputStream(rawimage);
        BufferedImage image = null;
        // create buffered image from received image
        try {
            image = ImageIO.read(rawImageStream);
        } catch (IOException e){
            Logging.log(LogService.LOG_ERROR,"edgeDetector: error
processing input image: %s", e.getMessage());
            return dm;
        }

        //apply detector to received image
        detector.setSourceImage(image);
        detector.process();
        // get resulting edge image
        BufferedImage edges = detector.getEdgesImage();
        // do some converting, image is ARGB, but need RGB for
jpg
```

```
        BufferedImage img = new BufferedImage(edges.getWidth(),
edges.getHeight(), BufferedImage.TYPE_INT_RGB);
        img.setRGB(0,  0, edges.getWidth(), edges.getHeight(),
edges.getRGB(0, 0, edges.getWidth(), edges.getHeight(), null, 0,
edges.getWidth()), 0, edges.getWidth());

        //convert image to byte array for insertion into digital
media object
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] edgeImageBytes = null;
        try {

            ImageIO.write(img, "jpg", baos);
            baos.flush();
            edgeImageBytes = baos.toByteArray();
            baos.close();
            // output image to file for testing purposes
            // ImageIO.write(img, "jpg", new
File(".\\WhatIsHappening.jpg"));

        } catch (IOException e){

            Logging.log(LogService.LOG_ERROR,"edgeDetector: error
converting edge profile image: %s", e.getMessage());
            return dm;

        }

        DigitalMedia dmEdge = new DigitalMedia(edgeImageBytes,
"image/jpg");

        return dmEdge;
    }

    /**
     * This class was created and released into the public domain
by Tom Gibara, and found from:
     * http://www.tomgibara.com/computer-vision/canny-edge-
detector
     *
     */

    public class CannyEdgeDetector {

        // statics

        private final static float GAUSSIAN_CUT_OFF = 0.005f;
        private final static float MAGNITUDE_SCALE = 100F;
        private final static float MAGNITUDE_LIMIT = 1000F;
        private final static int MAGNITUDE_MAX = (int)
(MAGNITUDE_SCALE * MAGNITUDE_LIMIT);

        // fields

        private int height;
        private int width;
        private int picsize;
```

```
        private int[] data;
        private int[] magnitude;
        private BufferedImage sourceImage;
        private BufferedImage edgesImage;

        private float gaussianKernelRadius;
        private float lowThreshold;
        private float highThreshold;
        private int gaussianKernelWidth;
        private boolean contrastNormalized;

        private float[] xConv;
        private float[] yConv;
        private float[] xGradient;
        private float[] yGradient;

        // constructors

        /**
         * Constructs a new detector with default parameters.
         */

        public CannyEdgeDetector() {
            lowThreshold = 2.5f;
            highThreshold = 7.5f;
            gaussianKernelRadius = 2f;
            gaussianKernelWidth = 16;
            contrastNormalized = false;
        }

        // accessors

        /**
         * The image that provides the luminance data used by
this detector to
         * generate edges.
         *
         * @return the source image, or null
         */

        public BufferedImage getSourceImage() {
            return sourceImage;
        }

        /**
         * Specifies the image that will provide the luminance
data in which edges
         * will be detected. A source image must be set before
the process method
         * is called.
         *
         * @param image a source of luminance data
         */

        public void setSourceImage(BufferedImage image) {
            sourceImage = image;
        }
```

```
        /**
         * Obtains an image containing the edges detected during
the last call to
         * the process method. The buffered image is an opaque
image of type
         * BufferedImage.TYPE_INT_ARGB in which edge pixels are
white and all other
         * pixels are black.
         *
         * @return an image containing the detected edges, or
null if the process
         * method has not yet been called.
         */

        public BufferedImage getEdgesImage() {
            return edgesImage;
        }

        /**
         * Sets the edges image. Calling this method will not
change the operation
         * of the edge detector in any way. It is intended to
provide a means by
         * which the memory referenced by the detector object may
be reduced.
         *
         * @param edgesImage expected (though not required) to be
null
         */

        public void setEdgesImage(BufferedImage edgesImage) {
            this.edgesImage = edgesImage;
        }

        /**
         * The low threshold for hysteresis. The default value is
2.5.
         *
         * @return the low hysteresis threshold
         */

        public float getLowThreshold() {
            return lowThreshold;
        }

        /**
         * Sets the low threshold for hysteresis. Suitable values
for this parameter
         * must be determined experimentally for each
application. It is nonsensical
         * (though not prohibited) for this value to exceed the
high threshold value.
         *
         * @param threshold a low hysteresis threshold
         */
```

```
        public void setLowThreshold(float threshold) {
            if (threshold < 0) throw new
IllegalArgumentException();
            lowThreshold = threshold;
        }

        /**
         * The high threshold for hysteresis. The default value
is 7.5.
         *
         * @return the high hysteresis threshold
         */

        public float getHighThreshold() {
            return highThreshold;
        }

        /**
         * Sets the high threshold for hysteresis. Suitable
values for this
         * parameter must be determined experimentally for each
application. It is
         * nonsensical (though not prohibited) for this value to
be less than the
         * low threshold value.
         *
         * @param threshold a high hysteresis threshold
         */

        public void setHighThreshold(float threshold) {
            if (threshold < 0) throw new
IllegalArgumentException();
            highThreshold = threshold;
        }

        /**
         * The number of pixels across which the Gaussian kernel
is applied.
         * The default value is 16.
         *
         * @return the radius of the convolution operation in
pixels
         */

        public int getGaussianKernelWidth() {
            return gaussianKernelWidth;
        }

        /**
         * The number of pixels across which the Gaussian kernel
is applied.
         * This implementation will reduce the radius if the
contribution of pixel
         * values is deemed negligable, so this is actually a
maximum radius.
         *
```

```
         * @param gaussianKernelWidth a radius for the
convolution operation in
         * pixels, at least 2.
         */

        public void setGaussianKernelWidth(int
gaussianKernelWidth) {
            if (gaussianKernelWidth < 2) throw new
IllegalArgumentException();
            this.gaussianKernelWidth = gaussianKernelWidth;
        }

        /**
         * The radius of the Gaussian convolution kernel used to
smooth the source
         * image prior to gradient calculation. The default value
is 16.
         *
         * @return the Gaussian kernel radius in pixels
         */

        public float getGaussianKernelRadius() {
            return gaussianKernelRadius;
        }

        /**
         * Sets the radius of the Gaussian convolution kernel
used to smooth the
         * source image prior to gradient calculation.
         *
         * @return a Gaussian kernel radius in pixels, must
exceed 0.1f.
         */

        public void setGaussianKernelRadius(float
gaussianKernelRadius) {
            if (gaussianKernelRadius < 0.1f) throw new
IllegalArgumentException();
            this.gaussianKernelRadius = gaussianKernelRadius;
        }

        /**
         * Whether the luminance data extracted from the source
image is normalized
         * by linearizing its histogram prior to edge extraction.
The default value
         * is false.
         *
         * @return whether the contrast is normalized
         */

        public boolean isContrastNormalized() {
            return contrastNormalized;
        }

        /**
         * Sets whether the contrast is normalized
```

```
         * @param contrastNormalized true if the contrast should
be normalized,
         * false otherwise
         */

        public void setContrastNormalized(boolean
contrastNormalized) {
            this.contrastNormalized = contrastNormalized;
        }

        // methods

        public void process() {
            width = sourceImage.getWidth();
            height = sourceImage.getHeight();
            picsize = width * height;
            initArrays();
            readLuminance();
            if (contrastNormalized) normalizeContrast();
            computeGradients(gaussianKernelRadius,
gaussianKernelWidth);
            int low = Math.round(lowThreshold * MAGNITUDE_SCALE);
            int high = Math.round( highThreshold *
MAGNITUDE_SCALE);
            performHysteresis(low, high);
            thresholdEdges();
            writeEdges(data);
        }

        // private utility methods

        private void initArrays() {
            if (data == null || picsize != data.length) {
                data = new int[picsize];
                magnitude = new int[picsize];

                xConv = new float[picsize];
                yConv = new float[picsize];
                xGradient = new float[picsize];
                yGradient = new float[picsize];
            }
        }

        //NOTE: The elements of the method below (specifically
the technique for
        //non-maximal suppression and the technique for gradient
computation)
        //are derived from an implementation posted in the
following forum (with the
        //clear intent of others using the code):
        //
http://forum.java.sun.com/thread.jspa?threadID=546211&start=45&ts
tart=0
        //My code effectively mimics the algorithm exhibited above.
        //Since I don't know the providence of the code that was
posted it is a
```

```
        //possibility (though I think a very remote one) that this
code violates
        //someone's intellectual property rights. If this concerns
you feel free to
        //contact me for an alternative, though less efficient,
implementation.


        private void computeGradients(float kernelRadius, int
kernelWidth) {

            //generate the gaussian convolution masks
            float kernel[] = new float[kernelWidth];
            float diffKernel[] = new float[kernelWidth];
            int kwidth;
            for (kwidth = 0; kwidth < kernelWidth; kwidth++) {
                float g1 = gaussian(kwidth, kernelRadius);
                if (g1 <= GAUSSIAN_CUT_OFF && kwidth >= 2) break;
                float g2 = gaussian(kwidth - 0.5f, kernelRadius);
                float g3 = gaussian(kwidth + 0.5f, kernelRadius);
                kernel[kwidth] = (g1 + g2 + g3) / 3f / (2f *
(float) Math.PI * kernelRadius * kernelRadius);
                diffKernel[kwidth] = g3 - g2;
            }

            int initX = kwidth - 1;
            int maxX = width - (kwidth - 1);
            int initY = width * (kwidth - 1);
            int maxY = width * (height - (kwidth - 1));

            //perform convolution in x and y directions
            for (int x = initX; x < maxX; x++) {
                for (int y = initY; y < maxY; y += width) {
                    int index = x + y;
                    float sumX = data[index] * kernel[0];
                    float sumY = sumX;
                    int xOffset = 1;
                    int yOffset = width;
                    for(; xOffset < kwidth ;) {
                        sumY += kernel[xOffset] * (data[index -
yOffset] + data[index + yOffset]);
                        sumX += kernel[xOffset] * (data[index -
xOffset] + data[index + xOffset]);
                        yOffset += width;
                        xOffset++;
                    }

                    yConv[index] = sumY;
                    xConv[index] = sumX;
                }

            }

            for (int x = initX; x < maxX; x++) {
                for (int y = initY; y < maxY; y += width) {
                    float sum = 0f;
                    int index = x + y;
```

```
                    for (int i = 1; i < kwidth; i++)
                        sum += diffKernel[i] * (yConv[index - i]
- yConv[index + i]);

                    xGradient[index] = sum;
                }

            }

            for (int x = kwidth; x < width - kwidth; x++) {
                for (int y = initY; y < maxY; y += width) {
                    float sum = 0.0f;
                    int index = x + y;
                    int yOffset = width;
                    for (int i = 1; i < kwidth; i++) {
                        sum += diffKernel[i] * (xConv[index -
yOffset] - xConv[index + yOffset]);
                        yOffset += width;
                    }

                    yGradient[index] = sum;
                }

            }

            initX = kwidth;
            maxX = width - kwidth;
            initY = width * kwidth;
            maxY = width * (height - kwidth);
            for (int x = initX; x < maxX; x++) {
                for (int y = initY; y < maxY; y += width) {
                    int index = x + y;
                    int indexN = index - width;
                    int indexS = index + width;
                    int indexW = index - 1;
                    int indexE = index + 1;
                    int indexNW = indexN - 1;
                    int indexNE = indexN + 1;
                    int indexSW = indexS - 1;
                    int indexSE = indexS + 1;

                    float xGrad = xGradient[index];
                    float yGrad = yGradient[index];
                    float gradMag = hypot(xGrad, yGrad);

                    //perform non-maximal supression
                    float nMag = hypot(xGradient[indexN],
yGradient[indexN]);
                    float sMag = hypot(xGradient[indexS],
yGradient[indexS]);
                    float wMag = hypot(xGradient[indexW],
yGradient[indexW]);
                    float eMag = hypot(xGradient[indexE],
yGradient[indexE]);
                    float neMag = hypot(xGradient[indexNE],
yGradient[indexNE]);
```

```
                    float seMag = hypot(xGradient[indexSE],
yGradient[indexSE]);
                    float swMag = hypot(xGradient[indexSW],
yGradient[indexSW]);
                    float nwMag = hypot(xGradient[indexNW],
yGradient[indexNW]);
                    float tmp;
                    /*
                     * An explanation of what's happening here,
for those who want
                     * to understand the source: This performs
the "non-maximal
                     * supression" phase of the Canny edge
detection in which we
                     * need to compare the gradient magnitude to
that in the
                     * direction of the gradient; only if the
value is a local
                     * maximum do we consider the point as an
edge candidate.
                     *
                     * We need to break the comparison into a
number of different
                     * cases depending on the gradient direction
so that the
                     * appropriate values can be used. To avoid
computing the
                     * gradient direction, we use two simple
comparisons: first we
                     * check that the partial derivatives have
the same sign (1)
                     * and then we check which is larger (2). As
a consequence, we
                     * have reduced the problem to one of four
identical cases that
                     * each test the central gradient magnitude
against the values at
                     * two points with 'identical support'; what
this means is that
                     * the geometry required to accurately
interpolate the magnitude
                     * of gradient function at those points has
an identical
                     * geometry (upto right-angled-
rotation/reflection).
                     *
                     * When comparing the central gradient to the
two interpolated
                     * values, we avoid performing any divisions
by multiplying both
                     * sides of each inequality by the greater of
the two partial
                     * derivatives. The common comparand is
stored in a temporary
                     * variable (3) and reused in the mirror case
(4).
                     *
```

```
                     */
                    if (xGrad * yGrad <= (float) 0 /*(1)*/
                        ? Math.abs(xGrad) >= Math.abs(yGrad)
/*(2)*/
                            ? (tmp = Math.abs(xGrad * gradMag))
>= Math.abs(yGrad * neMag - (xGrad + yGrad) * eMag) /*(3)*/
                                && tmp > Math.abs(yGrad * swMag -
(xGrad + yGrad) * wMag) /*(4)*/
                            : (tmp = Math.abs(yGrad * gradMag))
>= Math.abs(xGrad * neMag - (yGrad + xGrad) * nMag) /*(3)*/
                                && tmp > Math.abs(xGrad * swMag -
(yGrad + xGrad) * sMag) /*(4)*/
                        : Math.abs(xGrad) >= Math.abs(yGrad)
/*(2)*/
                            ? (tmp = Math.abs(xGrad * gradMag))
>= Math.abs(yGrad * seMag + (xGrad - yGrad) * eMag) /*(3)*/
                                && tmp > Math.abs(yGrad * nwMag +
(xGrad - yGrad) * wMag) /*(4)*/
                            : (tmp = Math.abs(yGrad * gradMag))
>= Math.abs(xGrad * seMag + (yGrad - xGrad) * sMag) /*(3)*/
                                && tmp > Math.abs(xGrad * nwMag +
(yGrad - xGrad) * nMag) /*(4)*/
                        ) {
                        magnitude[index] = gradMag >=
MAGNITUDE_LIMIT ? MAGNITUDE_MAX : (int) (MAGNITUDE_SCALE *
gradMag);
                        //NOTE: The orientation of the edge is
not employed by this
                        //implementation. It is a simple matter
to compute it at
                        //this point as: Math.atan2(yGrad,
xGrad);
                    } else {
                        magnitude[index] = 0;
                    }
                }
            }
        }

        //NOTE: It is quite feasible to replace the
implementation of this method
        //with one which only loosely approximates the hypot
function. I've tested
        //simple approximations such as Math.abs(x) + Math.abs(y)
and they work fine.
        private float hypot(float x, float y) {
            return (float) Math.hypot(x, y);
        }

        private float gaussian(float x, float sigma) {
            return (float) Math.exp(-(x * x) / (2f * sigma *
sigma));
        }

        private void performHysteresis(int low, int high) {
            //NOTE: this implementation reuses the data array to
store both
```

```
            //luminance data from the image, and edge intensity
from the processing.
            //This is done for memory efficiency, other
implementations may wish
            //to separate these functions.
            Arrays.fill(data, 0);

            int offset = 0;
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    if (data[offset] == 0 && magnitude[offset] >=
high) {
                        follow(x, y, offset, low);
                    }
                    offset++;
                }
            }
        }

        private void follow(int x1, int y1, int i1, int
threshold) {
            int x0 = x1 == 0 ? x1 : x1 - 1;
            int x2 = x1 == width - 1 ? x1 : x1 + 1;
            int y0 = y1 == 0 ? y1 : y1 - 1;
            int y2 = y1 == height -1 ? y1 : y1 + 1;

            data[i1] = magnitude[i1];
            for (int x = x0; x <= x2; x++) {
                for (int y = y0; y <= y2; y++) {
                    int i2 = x + y * width;
                    if ((y != y1 || x != x1)
                        && data[i2] == 0
                        && magnitude[i2] >= threshold) {
                        follow(x, y, i2, threshold);
                        return;
                    }
                }
            }
        }

        private void thresholdEdges() {
            for (int i = 0; i < picsize; i++) {
                data[i] = data[i] > 0 ? -1 : 0xff000000;
            }
        }

        private int luminance(float r, float g, float b) {
            return Math.round(0.299f * r + 0.587f * g + 0.114f *
b);
        }

        private void readLuminance() {
            int type = sourceImage.getType();
            if (type == BufferedImage.TYPE_INT_RGB || type ==
BufferedImage.TYPE_INT_ARGB) {
                int[] pixels = (int[])
sourceImage.getData().getDataElements(0, 0, width, height, null);
```

```
                for (int i = 0; i < picsize; i++) {
                    int p = pixels[i];
                    int r = (p & 0xff0000) >> 16;
                    int g = (p & 0xff00) >> 8;
                    int b = p & 0xff;
                    data[i] = luminance(r, g, b);
                }
            } else if (type == BufferedImage.TYPE_BYTE_GRAY) {
                byte[] pixels = (byte[])
sourceImage.getData().getDataElements(0, 0, width, height, null);
                for (int i = 0; i < picsize; i++) {
                    data[i] = (pixels[i] & 0xff);
                }
            } else if (type == BufferedImage.TYPE_USHORT_GRAY) {
                short[] pixels = (short[])
sourceImage.getData().getDataElements(0, 0, width, height, null);
                for (int i = 0; i < picsize; i++) {
                    data[i] = (pixels[i] & 0xffff) / 256;
                }
            } else if (type == BufferedImage.TYPE_3BYTE_BGR) {
                byte[] pixels = (byte[])
sourceImage.getData().getDataElements(0, 0, width, height, null);
                int offset = 0;
                for (int i = 0; i < picsize; i++) {
                    int b = pixels[offset++] & 0xff;
                    int g = pixels[offset++] & 0xff;
                    int r = pixels[offset++] & 0xff;
                    data[i] = luminance(r, g, b);
                }
            } else {
                throw new IllegalArgumentException("Unsupported
image type: " + type);
            }
        }

        private void normalizeContrast() {
            int[] histogram = new int[256];
            for (int i = 0; i < data.length; i++) {
                histogram[data[i]]++;
            }
            int[] remap = new int[256];
            int sum = 0;
            int j = 0;
            for (int i = 0; i < histogram.length; i++) {
                sum += histogram[i];
                int target = sum*255/picsize;
                for (int k = j+1; k <=target; k++) {
                    remap[k] = i;
                }
                j = target;
            }

            for (int i = 0; i < data.length; i++) {
                data[i] = remap[data[i]];
            }
        }
```

```
        private void writeEdges(int pixels[]) {
            //NOTE: There is currently no mechanism for obtaining
the edge data
            //in any other format other than an INT_ARGB type
BufferedImage.
            //This may be easily remedied by providing
alternative accessors.
            if (edgesImage == null) {
                edgesImage = new BufferedImage(width, height,
BufferedImage.TYPE_INT_ARGB);
            }
            edgesImage.getWritableTile(0, 0).setDataElements(0,
0, width, height, pixels);
        }

    }

}
```

# Appendix H. edgeDetectorConfigInterface.java

This appendix appears in its original form, without editorial change.

```
package mil.arl.alg.edgeDetector;

import aQute.bnd.annotation.metatype.Meta.AD;
import aQute.bnd.annotation.metatype.Meta.OCD;

@OCD(name = "edgeDetector plug-in") // <- tells BND this
interface provides ConfigurationAdmin data
public interface edgeDetectorConfigInterface {
    @AD(required=false, deflt = "true") // <- tells BND this is a
configuration attribute definition, and provides a default value
    boolean Run();
    @AD(required=false, deflt = "0.5")
    float lowThreshold();
    @AD(required=false, deflt = "1")
    float highThreshold();
}
```

# Appendix I. Open Standards for Unattended Sensors (OSUS) Plug-in Compliance Checklist

General points for verification of correct OSUS plug-in use:

- ☐ Is the plug-in of the proper type (asset, extension, etc.)?
- ☐ Does the plug-in follow a proper naming convention (mil.arl.example.ExampleAsset)?
- ☐ Is there a user's guide?
- ☐ Are all nonstandard dependencies packaged properly into the jar, or resources included?
- ☐ Does the documentation do any of the following:
  - ☐ Contain background on the plug-in's purpose, any underlying devices, and use cases?
  - ☐ Contain information on configuring the plug-in, including typical configurations?
  - ☐ Describe all configuration parameters?
  - ☐ Contain information on the dependencies?
  - ☐ Contain information on the built-in test?
  - ☐ Describe a test procedure for ensuring proper operation?
  - ☐ Describe the data the plug-in produces?
  - ☐ Describe the data the plug-in consumes?
  - ☐ Describe any error messages the plug-in could produce?
  - ☐ Describe the typical use cases of the plug-in?
  - ☐ Document the supported commands?
  - ☐ Have a list of capabilities?*
  - ☐ Contain a "quick start" guide?

Verification checks in the source code:

- ☐ Does the capabilities-xml accurately describe the plug-in's capabilities?*
- ☐ Are the export-package, include-resources, and private-package set properly in the bnd.bnd file?
- ☐ Are the initialize, activate, deactivate, and update methods handled properly?
- ☐ Do all status messages accurately reflect the requested status?
- ☐ Does the built-in test (PerformBIT) accurately check the system health?
- ☐ Are all data properly mapped to OSUS observations (i.e., no data in wrong fields)?
- ☐ Is there any specific code to prevent the plug-in from running on another operating system?
- ☐ Is proper logging used (use log.logging not System.out.print)?

* Asset plug-ins only.

If source-code distribution is provided:

- ☐ Does the code compile?
- ☐ Are all necessary source files included (configInterface, scanner, etc.)?
- ☐ Does the newly compiled plug-in boot and function without errors?

Verification checks from OSUS-Standard Graphical user interface (SG):

- ☐ Is the plug-in loaded into the system and activated?
- ☐ Is the plug-in accurately recognized by OSUS-SG?
- ☐ Are all of the necessary configuration parameters visible?
- ☐ Does the PerformBIT return accurate information?
- ☐ Does the activation and deactivation of the plug-in behave properly?
- ☐ Does the "captureData" button work properly?*
- ☐ Are configuration updates handles correctly?
- ☐ Are the commands handled properly?
- ☐ Do the OSUS observations contain all necessary, accurate, and relevant information?
- ☐ If the plug-in consumes data, are the time-consuming operations handled on another thread?
- ☐ If data have been mapped to an observation-reserved field:
  - ☐ Is there documentation describing the data and their format?
  - ☐ Is there justification that it could not be accurately placed into an existing field?

* Asset plug-ins only.

## List of Symbols, Abbreviations, and Acronyms

| | |
|---|---|
| ARGB | alpha–red–green–blue |
| ARL | US Army Research Laboratory |
| GUI | graphical user interface |
| IDE | Integrated Development Environment |
| OSGi | Open Source Gateway Initiative |
| OSUS | Open Standard for Unattended Sensors |
| PC | personal computer |
| RGB | red–green–blue |
| SDK | software development kit |
| SG | Standard GUI |
| UGS | unattended ground sensor |
| UUID | universally unique identifier |
| UVC | USB video class |
| XML | Extensible Markup Language |
| XSD | XML Schema Definition |

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIR ARL
IMAL HRA
RECORDS MGMT
RDRL DCL
TECH LIB

1 (PDF) GOVT PRINTG OFC
A MALHOTRA

1 (PDF) ARL
RDRL SES-A
J TYO